

Kladde

Specification and design documentation

Built from commit dfebfcf of 27 October 2026

Contents

1 kladde **16**
- 1.1 Project status . . . 16
- 1.2 The ambition . . . 17
- 1.3 Language-independent tooling . . . 17
- 1.4 The sections . . . 17
- 1.5 License . . . 18

2 Specification **19**
- 2.1 The layers . . . 19
- 2.2 The compatibility contract . . . 20
- 2.3 What is fixed and what is free . . . 20
- 2.4 Guarantees an implementation must provide . . . 20
- 2.5 Versioning . . . 21
- 2.6 Status . . . 21

3 File format **22**
- 3.1 Integer encodings and CRCs . . . 22
- 3.2 Pages . . . 22
- 3.3 Page framing . . . 22
  - 3.3.1 How much of the framing is load-bearing . . . 23
- 3.4 Header pages . . . 23
  - 3.4.1 The journal pointer . . . 25
  - 3.4.2 The private chain . . . 25
  - 3.4.3 The magic . . . 25
- 3.5 Epochs . . . 26
- 3.6 Versioning . . . 26
- 3.7 Concurrency . . . 27

4 Durability **28**
- 4.1 What is guaranteed . . . 28
- 4.2 Page states . . . 28
  - 4.2.1 The reuse rule . . . 29
  - 4.2.2 The truncation rule . . . 30
  - 4.2.3 Deallocating reusable pages . . . 30
- 4.3 The flush protocol . . . 30
  - 4.3.1 The two invariants . . . 31
  - 4.3.2 Why one `fsync` suffices . . . 31
- 4.4 Recovery / loading a kladde file . . . 32
  - 4.4.1 Case analysis for a power cut during flush `E` . . . 32
- 4.5 `fsync` failure is fatal . . . 32
- 4.6 Headroom . . . 33
  - 4.6.1 Detecting that the disk is full . . . 33
- 4.7 What this costs . . . 33

5 Span table **35**
- 5.1 The idea . . . 35

5.2 Logical level . . . 35
5.2.1 Entries . . . 35
5.2.2 Resolution . . . 36
5.2.3 Rules for writers . . . 36
5.3 Physical format . . . 36
5.3.1 Delta encoding . . . 37
5.3.2 A worked example . . . 37
5.3.3 Notes on the physical format . . . 37
5.4 Bounds . . . 38
5.5 Design directions . . . 38

6 The sequence **40**
6.1 Three chains . . . 40
6.2 One sequence, holding the root value's encoding . . . 40
6.3 What a valid sequence is . . . 41
6.4 Free space . . . 41
6.5 Reclamation is not part of this specification . . . 42
6.6 Partial loading, later . . . 42

7 Journal **43**
7.1 What the journal is . . . 43
7.2 Record kinds . . . 43
7.2.1 Cuts . . . 44
7.3 Transactions . . . 45
7.4 Ordering . . . 45
7.5 Encoding . . . 45
7.5.1 Records . . . 45
7.5.2 Seals . . . 46
7.5.3 External bytes . . . 46
7.5.4 Framing . . . 47
7.5.5 Pages . . . 48
7.5.6 The CRC chain . . . 48
7.6 Recovery . . . 49
7.7 The start of a session . . . 49
7.8 Folding . . . 49
7.8.1 Checkpoint versus commit . . . 50

8 Schema **51**
8.1 Why a file carries its schema . . . 51
8.2 The documents . . . 51
8.3 The guiding principle . . . 51
8.4 Scope . . . 52

9 Type descriptors **53**
9.1 Terminology . . . 53
9.2 The kinds . . . 53
9.2.1 Primitive . . . 53
9.2.2 Struct . . . 54
9.2.3 Enum . . . 54
9.2.4 Array . . . 55
9.2.5 Sequence . . . 55
9.2.6 Map . . . 55
9.2.7 Text . . . 55
9.2.8 Opaque . . . 55
9.2.9 Fixed . . . 56
9.3 Encodings . . . 56
9.3.1 A worked example . . . 57
9.4 Fixed-size types . . . 57
9.5 Rules of nesting . . . 57

9.6 References and the root . . . 57

10 Canonical encoding **59**
10.1 Encoding primitives . . . 59
10.2 Descriptor encoding . . . 59
10.3 Canonical order . . . 60
10.4 Table encoding . . . 60
10.5 Worked example . . . 60

11 Fingerprints **61**
11.1 The value . . . 61
11.2 Hash function . . . 61
11.3 What is and is not fingerprinted . . . 61
11.4 Cycles . . . 62
11.5 The computation, exactly . . . 62
11.5.1 Vectors . . . 62
11.6 Determinism . . . 63
11.7 As a fast path . . . 63

12 Evolution **64**
12.1 The levels . . . 64
12.2 What is settled . . . 64
12.3 What is TBD . . . 65
12.4 What was considered and declined . . . 66
12.5 Prior art . . . 66

13 Tooling **67**
13.1 The premise . . . 67
13.2 Level 1 — storage analysis . . . 67
13.3 Level 2 — schema inspection . . . 67
13.4 Level 3 — structural export . . . 68
13.5 A canonical textual representation . . . 68
13.6 Cross-implementation testing . . . 68

14 Conformance **69**
14.1 What conformance means . . . 69
14.2 Must . . . 69
14.3 May . . . 70
14.4 Must not . . . 70
14.5 The suite . . . 70
14.6 Open questions . . . 71

15 Drafts **72**
15.1 Current drafts . . . 72

16 Open issues **74**
16.1 Contradictions . . . 74
16.1.1 1. ~~The journal is two different things~~ — resolved . . . 74
16.1.2 2. ~~Journal pages cannot satisfy the page framing~~ — resolved . . . 74
16.1.3 3. The durability promise is stated unqualified where users read it . . . 74
16.1.4 4. Framing is per-record in one place and per-transaction in another . . . 75
16.1.5 5. `mentions` is claimed to have one reader and has at least two . . . 75
16.1.6 6. Reclamation is optional and also required . . . 75
16.1.7 7. Conversion is gone, and the flush still reasons about it . . . 75
16.1.8 8. “Five layers”, six rows . . . 75
16.1.9 9. Which “read” the complexity bound governs — narrowed, not closed . . . 76
16.1.10 10. Grouping is “free” but constrained . . . 76
16.2 Blockers for a first prototype . . . 76
16.2.1 1. ~~The journal’s on-file representation~~ — resolved . . . 76

16.2.2 2. ~~The byte encoding of journal records~~ — resolved . . . 76
16.2.3 3. Bootstrapping — partly described . . . 76
16.2.4 4. File extension is not covered by the durability argument . . . 76
16.2.5 5. The header's byte layout . . . 77
16.2.6 6. The pointer encoding inside allocations . . . 77
16.2.7 7. The in-memory address table . . . 77
16.2.8 8. The fold, plus its differential oracle . . . 77
16.2.9 9. Recording every mutation . . . 77
16.2.10 10. Freeing . . . 77
16.2.11 11. Choosing values for things that only need *a* value . . . 77
16.2.12 Deferrable, and why . . . 78
16.3 Ideas for later . . . 78

17 The Move operation **79**
17.1 It needs no new address-table statement . . . 79
17.2 It keeps the byte-referenced-once invariant . . . 79
17.3 What it buys . . . 79
17.4 Settled by the record's semantics . . . 80
17.5 Open questions . . . 80

18 Segments **81**
18.1 Why it is reachable at all . . . 81
18.2 What has already been decided with segments in mind . . . 81
18.3 What is not decided . . . 81

19 Cleaning by ripeness **82**
19.1 The problem . . . 82
19.1.1 A mixed page drains, then freezes . . . 82
19.1.2 Why the current design never cleans a frozen page . . . 82
19.1.3 What LFS did about it . . . 83
19.2 The proposal . . . 83
19.2.1 When waiting no longer pays . . . 83
19.2.2 A page holds a draining share and a static one . . . 84
19.2.3 Estimating how fast a page still drains . . . 85
19.2.4 Checking the fit in simulation . . . 88
19.2.5 Ranking pages by ripeness . . . 88
19.2.6 The budget, and the price . . . 90
19.2.7 Where survivors go . . . 90
19.3 What would change in Consolidation . . . 91
19.4 Ablation: without the logarithm . . . 92
19.5 Open questions . . . 93

20 Bayesian ripeness **94**
20.1 Executive summary . . . 94
20.2 The decision under a posterior . . . 95
20.2.1 The gain of cleaning now . . . 95
20.2.2 Three ways to decide . . . 95
20.3 One rate per page . . . 95
20.3.1 Losses counted in bytes, weighed in loss events . . . 95
20.3.2 The exact posterior is a Gamma distribution . . . 96
20.3.3 A rate that drifts keeps the posterior exact . . . 96
20.3.4 The single-rate draft is this posterior's mean . . . 96
20.3.5 What a page keeps . . . 97
20.3.6 Where a page's prior comes from . . . 97
20.3.7 The floor . . . 98
20.3.8 Deciding on one rate . . . 98
20.3.9 Waiting teaches, and the drafts' rule does not see it . . . 99
20.4 A draining share over a static one . . . 100
20.4.1 The cure model, per chunk . . . 100

20.4.2 The exact posterior is a mixture over the untouched chunks that drain 100
20.4.3 The prior replaces the test 101
20.4.4 A drifting rate 101
20.4.5 What a page keeps, and what a loss costs 101
20.4.6 Deciding on a mixture 102
20.4.7 How it relates to the draft's fit 102
20.5 Checking in simulation 102
20.5.1 Where a page's prior comes from, in simulation 104
20.6 What would change 104
20.7 Open questions 105

21 Sorting survivors by temperature **106**
21.1 The problem 106
21.2 The proposal 106
21.2.1 Three classes 106
21.2.2 The evidence 107
21.2.3 Where each class goes 107
21.2.4 Within a class, key order 108
21.2.5 What it costs 108
21.2.6 What it gives the estimates 109
21.2.7 What to expect on kladde-bench 109
21.3 Alternatives 109
21.4 What would change 110
21.5 Open questions 110

22 Variable-size values **111**
22.1 The problem 111
22.2 The proposal 112
22.2.1 Encodings: fixed and packed 112
22.2.2 Pointers, and what they point to 114
22.2.3 Which library a type belongs to, later 114
22.2.4 Small values 115
22.2.5 Who chooses, and how the choice nests 116
22.2.6 In the schema 116
22.3 What a mutation costs 117
22.3.1 The size stays the same 117
22.3.2 The size changes 117
22.3.3 In memory 118
22.4 Why it fits kladde 118
22.5 What else it touches 118
22.5.1 Containers 118
22.5.2 Evolution 119
22.5.3 Tooling 119
22.5.4 kladde-rs 119
22.6 Stages 120
22.7 Left for later: floats as decimals 120
22.7.1 What it would save 120
22.7.2 What others do 121
22.7.3 What adopting it would take 121
22.8 Alternatives 121
22.8.1 A durable `Box` for large variants 122
22.8.2 Struct of arrays, in the application 122
22.8.3 Compressing data pages 122
22.8.4 `Zero` statements for padding 122
22.9 Open questions 122

23 Sparse leaves, and consolidation from the statements up **123**
23.1 The finding 123
23.1.1 Before the fill floor went 124

23.1.2 Before the newest size won . . . 125
23.2 Why leaves stayed empty . . . 125
23.2.1 The budget loop held a table offer to the data pages' fill floor . . . 125
23.2.2 Leaves were kept alive for the size alone . . . 126
23.2.3 The cut's fillers refused every leaf that named an id the flush touched . . . 126
23.2.4 What the check guarded against . . . 127
23.2.5 Dead statements cannot block a rewrite . . . 127
23.2.6 Compaction mode collected them, in bursts . . . 127
23.2.7 Data pages are not hit the same way . . . 127
23.3 The underlying error . . . 128
23.4 Possible solutions, from the statements up . . . 128
23.4.1 What a writer must guarantee . . . 128
23.4.2 Task 1: choose the victims . . . 128
23.4.3 Task 2: combine the victims with the fold and the header . . . 129
23.4.4 Task 3: write the result . . . 129
23.4.5 A spec change that would make it simpler still: statements keep their epoch . . . 130
23.4.6 What this would remove . . . 130
23.4.7 Open questions . . . 130

24 Splices without restating the tail **131**
24.1 What a splice costs today . . . 131
24.2 What every design has to answer . . . 132
24.3 The designs . . . 133
24.3.1 A. Offsets, restated or merged: today . . . 133
24.3.2 B. Chunked containers: keep allocations short . . . 133
24.3.3 C. Spans placed by offset . . . 134
24.3.4 D. Linked spans over backings: the recommendation . . . 134
24.3.5 E. Spans keyed by dense positions . . . 134
24.3.6 F. A tree of spans . . . 135
24.3.7 G. Shift statements . . . 135
24.4 The recommended design in detail . . . 135
24.4.1 Statements . . . 135
24.4.2 Resolution . . . 136
24.4.3 Rules for writers . . . 136
24.4.4 Encoding . . . 136
24.4.5 Open and memory . . . 136
24.4.6 Flushing . . . 137
24.4.7 Compacting spans . . . 137
24.4.8 What it changes elsewhere . . . 137
24.5 Measured . . . 137
24.5.1 How . . . 138
24.5.2 What the flushes state . . . 138
24.5.3 What accumulates . . . 139
24.6 What the literature does . . . 139
24.7 Open questions . . . 140

25 A C++ implementation **141**
25.1 What drives the design . . . 141
25.2 Three layers . . . 142
25.3 The shape of the API . . . 142
25.4 The analog of `#[derive(Durable)]` . . . 143
25.4.1 The options . . . 143
25.4.2 The macro . . . 143
25.4.3 Reflection . . . 144
25.4.4 Generics, and what is durable . . . 145
25.5 Which C++ types are durable . . . 145
25.6 Enums . . . 147
25.6.1 `enum class`: variants without fields . . . 147
25.6.2 `std::variant`: a tagged union with value semantics . . . 147

25.6.3 Class hierarchies: a closed set of classes, held polymorphically . . . 148
25.6.4 Which to use . . . 148
25.7 Handles: where a value lies . . . 149
25.7.1 Short-lived handles . . . 149
25.7.2 Long-lived references . . . 149
25.8 Errors, transactions and exception safety . . . 150
25.9 Flushing on an interactive thread . . . 151
25.10 Files and platforms . . . 152
25.11 The Inkscape demo . . . 152
25.12 Changes to the specification . . . 153
25.12.1 Variants that hold one value . . . 153
25.12.2 Maps as a kind of their own . . . 154
25.12.3 A flush off the caller's thread . . . 154
25.12.4 Cuts that outlive their transaction . . . 155
25.12.5 Smaller points . . . 156
25.13 Building, packaging and testing . . . 157
25.14 Order of work . . . 157
25.15 Open questions . . . 158

26 Evaluation **160**
26.1 The evaluations . . . 160

27 Benchmarks and tests **162**
27.1 At a glance . . . 162
27.2 The tests . . . 163
27.3 The tables . . . 164
27.3.1 Reading a table . . . 166
27.4 `kladde-bench`: synthetic workloads . . . 166
27.5 `text-bench`: typing into one text . . . 167
27.6 `svg-bench`: a drawing editor . . . 168
27.7 `svg-roundtrip`: a corpus through kladde . . . 169
27.8 Tools in this repository . . . 170
27.8.1 The figures . . . 170
27.8.2 The ripeness simulation . . . 170
27.8.3 Checks of this repository . . . 170
27.8.4 Continuous integration . . . 171
27.9 A measurement kept with its evaluation . . . 171

28 Consolidation under load **172**
28.1 What was measured . . . 172
28.2 Space . . . 173
28.3 The churn floor sets the fill, not the target . . . 174
28.4 Writes . . . 176
28.5 Mass frees . . . 176
28.6 The description . . . 177
28.7 Time . . . 178
28.8 The typed layers . . . 179
28.9 Defects the benchmarks found . . . 179
28.10 What to change . . . 179
28.11 Reproducing . . . 180

29 Cleaning by ripeness **181**
29.1 What was measured . . . 181
29.2 The trade-off . . . 182
29.3 The controller . . . 183
29.4 At the default settings . . . 184
29.5 Hot, cool, and cold allocations on shared pages . . . 186
29.6 Without the logarithm . . . 186
29.7 What it costs . . . 188

29.8 What to change in the draft . . . 189
29.9 Reproducing . . . 189

30 Ripeness with a static share **191**
30.1 What was measured . . . 191
30.2 How often pages earn a static share . . . 191
30.3 Cleaning . . . 192
30.4 What it costs . . . 194
30.5 What to change in the draft . . . 195
30.6 Reproducing . . . 195

31 Bayesian ripeness **196**
31.1 What was measured . . . 196
31.2 Cleaning . . . 197
31.3 What the posterior calls static . . . 198
31.4 The empirical prior . . . 199
31.5 What it costs . . . 201
31.6 What to change in the draft . . . 202
31.7 Reproducing . . . 202

32 Cleaning edited drawings and texts **204**
32.1 Executive summary . . . 204
32.2 What was measured . . . 205
32.3 Live pages . . . 207
32.4 Leaves . . . 210
32.5 The trade-off between space and writes . . . 210
32.6 Files . . . 212
32.7 Writes . . . 214
32.8 Times . . . 215
32.9 The sweep in numbers . . . 216

33 Ripeness with main's free filling **219**
33.1 Executive summary . . . 219
33.2 What changed . . . 219
33.3 What was measured . . . 220
33.4 Free filling . . . 221
33.5 The trade-off between space and writes . . . 221
33.6 The regime . . . 223
33.7 Live pages and fill . . . 224
33.8 Files . . . 226
33.9 Writes . . . 227
33.10 The sweep in numbers . . . 228

34 The file as one sequence, measured **230**
34.1 Executive summary . . . 230
34.2 What was measured . . . 231
34.3 The trade-off between space and writes . . . 232
34.4 The live size: the encoding . . . 234
34.5 Live pages . . . 234
34.6 The span policy . . . 235
34.7 Files . . . 236
34.8 Writes . . . 237
34.9 Times . . . 238
34.10 The sweep in numbers . . . 239

35 Porting kladde to C++ and into Inkscape **241**
35.1 What was built . . . 241
35.2 Two implementations, one format . . . 241
35.3 What the specification left open . . . 242

35.4 What C++ made harder . . . 242
35.4.1 Handles that nothing checks . . . 242
35.4.2 Describing types without a derive . . . 243
35.4.3 Long-lived handles . . . 243
35.5 What Inkscape needed . . . 243
35.6 Measurements . . . 244
35.6.1 What the typed schema took . . . 245
35.6.2 The journal's budget . . . 246
35.6.3 Ablation: attribute values as text . . . 247
35.6.4 In the GUI . . . 248
35.7 Bugs the port found in kladde-rs . . . 248
35.8 Reproducing . . . 248

36 Related work **250**
36.1 Systems that address related problems . . . 250
36.1.1 Saving the whole document . . . 251
36.1.2 A journal beside a checkpoint of the whole state . . . 251
36.1.3 Embedded databases: SQLite and its kin . . . 251
36.1.4 Object stores . . . 252
36.1.5 Collaborative documents: CRDTs . . . 252
36.1.6 Durable containers in persistent memory . . . 253
36.1.7 What they offer that kladde does not . . . 253
36.2 Systems with related internal parts . . . 253
36.2.1 The encoding: Avro's, nearly . . . 254
36.2.2 The sequence on file: a piece table, as Word's fast save kept one . . . 254
36.2.3 Storage: a log-structured file system for one file . . . 255
36.2.4 State and durability: a memory image with incremental checkpoints . . . 255
36.2.5 Moving values: Cap'n Proto's orphans . . . 256
36.2.6 Order by links: sequence CRDTs . . . 256
36.2.7 Where the redesigns have led . . . 256

37 Implementation notes **257**
37.1 Where the implementation falls short . . . 257
37.2 Choices kladde-docs leaves open . . . 259

38 Implementation **260**
38.1 The documents . . . 260
38.2 The shape of an implementation . . . 260
38.3 Overhead and work independent of size . . . 261
38.3.1 Where the bytes of the file go . . . 261
38.3.2 Each term, and what keeping it costs . . . 261
38.3.3 A worked example . . . 262
38.4 The mirror . . . 263

39 In-memory state **264**
39.1 1. The span table . . . 264
39.2 2. The rope . . . 264
39.3 3. The page table . . . 265
39.4 4. The reverse index . . . 265
39.5 5. The span id allocator . . . 265
39.6 Liveness is one bit . . . 266
39.7 Heat . . . 266
39.8 During a flush . . . 266
39.9 Epochs belong to pages, not to entries . . . 266
39.10 Memory for very large files . . . 266

40 Opening a file **267**
40.1 The steps . . . 267
40.2 Replaying through a flush . . . 267

40.3 The seeds of the rotating walk and of heat . . . 268

41 Values and their positions **269**
41.1 A value is a region . . . 269
41.2 Positions belong to containers . . . 270
41.3 What a container keeps to find its children . . . 270
41.4 What a handle costs . . . 271
41.4.1 Handles made together . . . 271
41.4.2 Indexes weighted by size . . . 272
41.5 Handles that outlive edits elsewhere . . . 272
41.6 The built-in containers . . . 273
41.7 Moving a value between containers . . . 273

42 Write-phase state **275**
42.1 The log is the sole authority . . . 275
42.2 Two derived structures . . . 275
42.3 After a flush, and after a rollback . . . 275

43 The flush **277**
43.1 The fold . . . 277
43.2 Mapping pieces to spans . . . 278
43.2.1 Payload pages . . . 278
43.2.2 Overwrites: split or rewrite . . . 278
43.2.3 Zeros . . . 279
43.3 The span policy . . . 279
43.4 The `Inline` threshold . . . 279
43.5 Placement . . . 280
43.5.1 Cutting content across pages . . . 280
43.5.2 Page contiguity buys nothing . . . 280
43.5.3 Where the pages come from . . . 280
43.6 The header as write buffer . . . 281
43.7 The shape of the tree . . . 281
43.8 The cut . . . 281
43.9 Correctness: the differential oracle . . . 281
43.10 Walk-through of an epoch . . . 282

44 Consolidation **283**
44.1 The high-level picture . . . 283
44.1.1 Three mechanisms . . . 283
44.2 Evacuation . . . 284
44.2.1 Victim selection for data pages . . . 284
44.2.2 Scoring a data page . . . 284
44.2.3 Victims are pulled one at a time . . . 285
44.3 The page rewrite . . . 285
44.3.1 Unlinking an emptied page . . . 285
44.4 Merging spans rides the rotating walk . . . 286
44.5 How a flush and consolidation compose . . . 286
44.5.1 Mandatory and elastic . . . 286
44.5.2 Slack is cheaper than a split — up to a point . . . 287
44.5.3 Packing in sequence order, with look-ahead . . . 287
44.6 The budget . . . 288
44.6.1 The structures behind the ranking . . . 288
44.6.2 The budget loop . . . 288
44.6.3 The complexity this incurs . . . 290
44.7 Two forms of consolidation . . . 290
44.8 Outer fragmentation . . . 290
44.8.1 Lowest page first . . . 290
44.8.2 Truncation . . . 290
44.8.3 Compaction mode . . . 290

44.9 Pacing . . . 291
44.10 The bound this buys . . . 291
44.11 Constants . . . 291

45 Consolidator state **292**
45.1 Why it exists . . . 292
45.2 What it would hold . . . 292
45.3 Writing it . . . 293
45.4 Checking it . . . 293

46 Transactions and batches **294**
46.1 Three mechanisms over one journal . . . 294
46.1.1 Recommendations . . . 295
46.2 Nesting . . . 295
46.3 One buffer and three cursors . . . 295
46.3.1 Transitions in detail . . . 297
46.4 Flush triggers . . . 298
46.4.1 The journal's budget . . . 299
46.4.2 Payload pages . . . 299
46.4.3 Why `ready` always fits . . . 300
46.4.4 Why flush first . . . 300
46.4.5 Consequence for write-phase state . . . 300
46.5 Checkpoint versus commit . . . 300
46.6 A flush off the caller's thread . . . 301
46.7 Open questions . . . 301

47 kladde-rs **302**
47.1 Two audiences . . . 302
47.2 What it is . . . 302
47.3 The central problem . . . 303
47.4 The documents . . . 303
47.5 Status . . . 303

48 Tutorial **304**
48.1 Is kladde the right tool? . . . 304
48.2 Status warning . . . 304

49 Getting started **305**
49.1 The mental model . . . 305
49.2 Your first durable value . . . 305
49.3 Why the `Mut` exists . . . 306
49.4 Nesting . . . 306

50 The built-in containers **307**
50.1 A default library, not a privileged one . . . 307
50.2 `String` . . . 307
50.3 `Vec<T>` and `DurableVec<T>` . . . 307
50.4 `DurableMap<K, V>` . . . 308
50.5 `DurableBlob<T>` . . . 309
50.6 What is missing . . . 309
50.7 Choosing . . . 309

51 Deriving your own types **310**
51.1 What to import . . . 310
51.2 Structs . . . 311
51.3 Fixed-size values . . . 311
51.4 Newtypes . . . 312
51.5 Enums . . . 312
51.6 Trait objects . . . 313

51.7 Generics . . . 314
51.8 What the derive needs . . . 314
51.9 Non-persisted fields . . . 314
51.10 Encoding stability . . . 315

52 Compared to serde **316**
52.1 The core difference . . . 316
52.2 What serde does better . . . 316
52.3 What kladde does better . . . 317
52.4 They are not exclusive . . . 317
52.5 Choosing . . . 317
52.6 An honest caveat . . . 317

53 Durability and flushing **318**
53.1 The two-stage model . . . 318
53.2 What a crash costs . . . 318
53.3 Flushing . . . 318
53.4 When the disk fails . . . 319
53.5 Transactions . . . 319
53.6 Long runs of mutations . . . 320

54 Implementing Durable by hand **321**
54.1 When you need one . . . 321
54.2 The trait . . . 321
54.3 What to import . . . 322
54.4 A complete example . . . 322
54.5 A value whose size changes . . . 324
54.6 Your descriptor . . . 324
54.7 The obligations you are taking on . . . 324

55 Crate layout **326**
55.1 The crates . . . 326
55.2 The parts . . . 328
55.2.1 `kladde-store` — the storage layer . . . 328
55.2.2 `kladde-persist` — the serialization layer . . . 328
55.2.3 `kladde-schema` — type descriptors . . . 328
55.2.4 `kladde-types` and `kladde-derive` — the type vocabulary . . . 328
55.2.5 `kladde` — the entry point . . . 328
55.3 The seam . . . 328
55.4 Cross-cutting concerns . . . 329

56 Durable and Mut **330**
56.1 `Durable` . . . 330
56.1.1 `FIXED_SIZE` . . . 330
56.1.2 `encode` and `decode` . . . 331
56.1.3 `Bounded` and `Fixed<T>` . . . 331
56.2 Regions . . . 331
56.3 `Mut` . . . 332
56.3.1 Mutations return a `Result` . . . 332
56.3.2 `Mut`s are generated per type . . . 332
56.4 Why an automatic flush cannot tear a mutation . . . 332
56.5 The chain, end to end . . . 333
56.6 What this costs . . . 333

57 Containers **334**
57.1 `Vec<T>` and `DurableVec<T>` . . . 334
57.2 `String` . . . 335
57.3 `DurableMap<K, V>` . . . 335
57.4 `DurableBlob<T>` . . . 335

57.5 Cuts . . . 335
57.6 What is missing . . . 336

58 The derive macro **337**
58.1 What it generates . . . 337
58.2 The aggregate records nothing of its own . . . 338
58.3 Fixed forms . . . 338
58.4 Primitives are `Durable` too . . . 338
58.5 Enums . . . 338
58.6 Trait objects, as an enum . . . 339
58.7 Generics and transparency . . . 340
58.8 Path resolution . . . 340
58.9 What is missing . . . 340
58.10 Testing compile failures . . . 341

59 Transactions **342**
59.1 RAII types . . . 342
59.2 Behaviour on unwind . . . 342
59.3 Open questions . . . 343

60 Schema binding **344**
60.1 Two layers . . . 344
60.2 The binding . . . 344
60.3 What a type declares . . . 344
60.4 At create and at open . . . 345
60.5 Design invariants . . . 345

61 The store **346**
61.1 The interface . . . 346
61.2 Interior mutability . . . 346
61.3 Storage . . . 346
61.4 The mirror . . . 347
61.5 The in-memory structures . . . 347
61.5.1 The span table . . . 347
61.5.2 The treap . . . 347
61.5.3 Maps keyed by id or page . . . 347
61.5.4 The journal buffer . . . 347
61.6 Where the integer widths come from . . . 347

62 kladde-cpp **349**
62.1 What it is . . . 349
62.2 Three layers . . . 349
62.3 Which C++ types are durable . . . 350
62.4 Handles . . . 351
62.5 Errors . . . 351
62.6 Flushing . . . 351
62.7 Inside the library . . . 351
62.8 Where kladde-cpp departs from the draft . . . 351

63 Superseded **352**
63.1 What was replaced, and by what . . . 352
63.2 A note on measurement . . . 353

64 The relocatable heap **354**
64.1 What it was . . . 354
64.2 The trait . . . 354
64.2.1 The heap owned the id table . . . 354
64.2.2 Ids came from outside . . . 355
64.2.3 A step was a byte range, not an allocation . . . 355

64.2.4 `budget` was a ranking input, not a cap . . . 355
64.3 Sizedness . . . 355
64.4 Errors and composition . . . 356
64.5 Placement . . . 356
64.6 `resize` could ask for two moves . . . 356

65 Incremental compaction **357**
65.1 The constraints . . . 357
65.2 The potential . . . 357
65.3 The candidate shapes . . . 358
65.4 The evacuation index . . . 358
65.4.1 The lesson worth keeping . . . 359
65.5 Pathologies . . . 359
65.5.1 The stance . . . 359
65.5.2 The stress test . . . 359
65.5.3 Remedy 1: weighting by class . . . 359
65.5.4 Remedy 2: the lift . . . 360
65.6 Measurement methodology . . . 360
65.7 What was still open when it was retired . . . 360

66 The in-place flush **362**
66.1 What it was, and what it needed . . . 362
66.2 Why it went . . . 362
66.3 What kladde does differently from those systems . . . 363
66.4 What was given up . . . 363

67 Whole-allocation table entries **364**
67.1 What it was . . . 364
67.2 Why per-chunk statements were rejected first . . . 364
67.3 The mechanisms it needed, and no longer does . . . 365
67.3.1 Extents . . . 365
67.3.2 Spilling . . . 365
67.3.3 A separate index . . . 365
67.3.4 Tombstone lifetime . . . 365
67.4 What else survived . . . 366

68 Sizes from extents **367**
68.1 Why it was abandoned . . . 367

69 Allocations and the address table **368**
69.1 What it was . . . 368
69.2 Why it was replaced . . . 369
69.3 Compared with the present design . . . 369
69.4 The vocabulary that went with it . . . 370

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.

Video: This fork of Inkscape needs no save button. It uses kladde's durable data structures.

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 bulk part holds the root value's encoding as one byte sequence, kept on copy-on-write pages as a chain of spans that each show a run of its bytes and name the span that follows. An edit anywhere — a keystroke in a long text, a shape moved to another group — restates the few spans where it cuts the chain and nothing behind them, so the file stays within a constant factor of its content at a constant cost per edit, however large it grows.

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 the reference implementations.

There are two reference implementations, kladde-rs in Rust and kladde-cpp in C++, and neither is *production ready*. Their purpose at the moment is to allow rapid experimentation on new *policies* (non-normative decisions that an implementation may make without breaking interoperability), and to test that the specification is precise enough for two independent implementations to read each other's files. This is why the reference implementations are currently limited to these two languages and why they are 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 and kladde-cpp second, with `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 `Mut` 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 kladde file's content is one byte sequence, the encoding of the application's root value with every value inline in its parent, and the file carries the descriptors of its types. So a generic tool can measure the file's garbage and compact it without interpreting a byte of application data, report what types the file contains, and decode its whole content from those descriptors 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
Drafts	half-baked ideas in progress, not yet held to the separation the other sections observe	the authors
Evaluation	measurements of kladde-rs on realistic workloads, what they say about the design, and what porting it to C++ found	anyone judging the design or choosing its constants

section	what it holds	who it is for
Related work	what applications use in kladde's place today, where the design comes from, and where it parts company	potential users exploring whether kladde is the right choice; contributors researching where ideas came from
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, `Mut` handles, macros, and the Rust-specific half of the algorithms	Rust users, and Rust contributors
kladde-cpp	everything specific to the C++ implementation: its layers, which C++ types are durable, its handles and its error handling	C++ users, and C++ contributors
Superseded	designs that were worked out and then rejected, kept for the reasoning	anyone tempted to re-propose one of them

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 reimplementation 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 six 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
Span table	the entries that describe the content as chains of spans, and how they are resolved	Span table
The sequence	what the chains hold: the root value's encoding, the schema, and the private state	The sequence
Journal	the record set for durable mutation of the sequence, transactions, framing, and recovery	Journal
Schema	how a value type is encoded, how its descriptor is serialized and fingerprinted, and how it evolves	Schema

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

In one paragraph: a kladde file is a grid of pages, committed copy-on-write through two alternating header pages. Its content is one byte sequence, the encoding of the application's root value, with every value inline in its parent, and no pointers. On file, the sequence is a chain of spans, each described by an entry in a table page that shows a run of its bytes and names the span that follows. Every mutation appends a record on positions in the sequence to a journal before it returns, and a flush periodically folds the journal into new table and data pages and commits them with one `fsync` and one header write.

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 sequence 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 span-table entry forms and the rules that resolve them;
- the journal record encoding and the rules for recovering a torn tail;
- the encoding of every value, 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, how it cuts content across pages, and how it cuts it into spans;
- **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 five 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 — span ids, the sequence's length, 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
a splice, overwrite or move, on file	a constant number of span entries, wherever it lands and however long the sequence
a mutation, in the store	`O(log k)` for `k` spans, besides the bytes it writes

operation	required
a flush	`O(r · log k)` for `r` records folded and the spans stated, plus the pages it writes
opening a file	`O(entries + n)` for the entries of the live table pages and a sequence of `n` bytes
a non-mutating operation once the file is loaded	no overhead over a comparable non-persisted data type

The live pages can be held at a constant multiple of the sequence's length, at a constant cost in bytes written per modification: the reference algorithms show how. 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.

Replacing a whole value records only what changed. When an application replaces a whole Text, or a whole Opaque value without a fixed size, with a new one, the records appended rewrite at most the smallest range of the old content that holds every changed byte, and, for an Opaque whose length changes, its length prefix. The implementation may rewrite the prefix and that range as one range, together with what lies between them, where it judges that cheaper — when the change lies near the start, say. Whether it goes further and finds several changed ranges far apart is its own choice.

Why. An application often sets a whole string where only part of it changed, because that is the easy thing to write: a drawing program that writes a path's whole `d` attribute for every step of a dragged control point, or a form that stores its whole text at every keystroke. Comparing the old content with the new costs a pass over memory the application has just written, and turns a rewrite of the whole value into a splice of a few bytes; guaranteeing it lets applications write the easy code without paying for it in the journal.

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 two reference implementations, in Rust and C++. The page, durability, span-table and sequence layers are settled in design and worked out in detail, and kladde-rs and kladde-cpp implement them byte for byte as stated, but nothing is 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 payload page is the one whole page written again while live: to append bytes, as the journal's last page is appended to, while its journal is the one being appended to. 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`	0 unless it is a header page	only present in header pages
`kind`	1 byte	one of `Table` (`0x01`) or `Data` (`0x02`). For header pages: always `Table` (`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 `Table` 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 a reference implementation 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
`sequence_head`	4 bytes	the id of the first span of the sequence, the root value's encoding, or `0` if the sequence is empty
`schema_head`	4 bytes	the id of the first span of the schema chain, which holds the descriptor table, or `0` if it is empty
`private_head`	4 bytes	the id of the first span of the private chain, in which an implementation keeps state of its own from one session to the next, or `0` if it is empty. See The private chain.
`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` field is 48 bytes, at offsets 0, 11, 12, 14, 16, 20, 24, 28 and 44 in the order above, so the page's `kind` sits at offset 48 and a header page's `content` at offset 59.

The header page **is** the root span-table page, so the span 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 alternatingly: the flush with epoch `E` writes header slot `E mod 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 private chain

`private_head` names a chain 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 an implementation's consolidator could keep there.

Four rules govern it:

- **Nothing in it affects what the file contains.** A reader that ignores the chain decodes the same sequence and the same schema; the canonical textual representation of a file leaves it out.
- **In every other respect it is a chain like the others.** Its spans obey the span table's rules, and no span belongs to two chains.
- **Any writer may keep it, replace it, or clear the field.** A writer that keeps it keeps its bytes, whether or not it understands them; it may restate or merge the chain's spans as it would any others.
- **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 `xxd | head` or `strings` 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 `type file` on Windows stops here instead of spewing the whole file

bytes	purpose
`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 the entries of one span** across table pages: the entry in the page of highest epoch describes the span, which is what lets a flush restate a span without rewriting the page that held its older entry (Span 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 `≤ 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 how to decode the new primitive, and so where any value after it begins. 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 copy-on-write commit is 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, and the payload pages its transactions name, 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, by whatever means its language makes idiomatic, and refuses further work — 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.

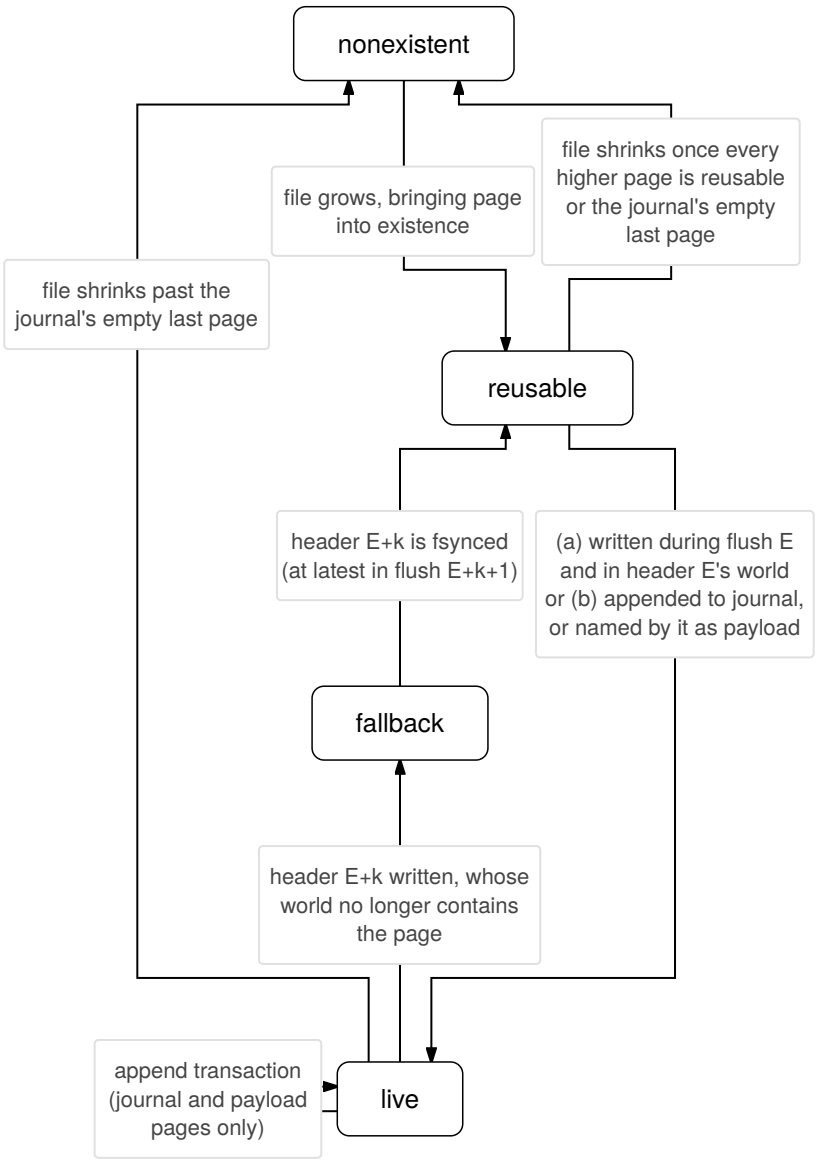
4.2 Page states

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

Define the **world** of a header as everything reachable from it: its span-table pages, the data pages that reached spans show bytes of, every page of the *next* epoch's **journal**, of which the header names only the first page, the rest reached from it, and every **payload page** that a complete transaction of that journal names. 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 `fsync`ed yet;
- **reusable** — in neither on-disk header's world, or only in the older header's world after the newer header has been `fsync`ed.



4.2.1 The reuse rule

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

The writes between flushes keep to the same rule: the journal and its payload pages take reusable pages or extend the file, and write again only pages of the journal being appended to.

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

The journal of epoch `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** fsynced individually.

Flush `E` then proceeds:

1. **Fold** journal `E` in memory into the new content of the sequence, as runs of bytes already in the file — in the committed sequence's data pages, or in the journal's payload pages of epoch `E` — and new bytes.
2. **Choose target pages**: reusable pages, else grow the file.
3. **Write data pages** for the new bytes; the payload pages that step 1 keeps join the header's world as data pages, written already. 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 span-table pages**: the entries of every span that steps 1–4 created or changed, as the reach rule requires and as the writer chooses beyond it.
6. **Choose a reusable start page for the journal of epoch `E + 1`.** That page needs no write, except in a session's first flush: that journal'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 journal 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 mod 2`: epoch `E`, the root span-table page's content, the three chains' first spans, the start page of the journal of epoch `E + 1`, 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.

A flush may seal its journal first: before step 1 it chooses the start page of the journal of epoch `E + 1` (step 6, moved forward), writes it if it is a session's first flush, followed by an `fsync` (why), and appends a seal naming it to journal `E`. Steps 1–8 then fold journal `E` up to the seal, while new transactions are appended to journal `E + 1`; the pages of journal `E + 1` and the pages steps 2–5 write are disjoint, since both come from the reusable pages and each takes its own. Invariant I1 is unchanged: the header names journal `E + 1`, which the `fsync` of step 7 need not cover, and recovery from header `E − 1` reaches it through the seal, so a transaction appended to it survives a crash of the application whichever header governs. Flush `E + 1` begins only once step 8 of flush `E` is done. 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 it names and that journal's payload pages, 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 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 the journal of epoch `E + 1`, which is still empty when `fsync E` runs, and so are the payload pages its transactions will name. It needs no protection from I1: it validates itself, transaction by transaction, through the chained epoch-salted CRCs, and its payload pages reference by reference, through the checksums its records carry. A payload page that flush `E` takes into its header's world was written before its transaction was appended to journal `E`, so `fsync E` covers it.

Consequence: a header found on disk implies that everything in its world except the journal 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 `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 `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 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* span table and walking *its* chains 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. Both reference implementations use 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 `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 `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 `E`, whose first page it names. Recovery lands on state `E − 1` plus any valid prefix of the (possibly torn) journal `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 `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 entries 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.

Both reference implementations therefore reserve 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 / τ`.
- **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 + span table + header), though a flush whose new bytes are all inline and whose entries fit in the header writes the header alone 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

Span table

The span table describes the file's content as chains of spans: each span shows a run of bytes and names the span that follows it. 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 file's content is three byte strings, each kept as a chain of spans: a span shows a run of bytes — bytes in a data page, bytes in its entry itself, or zeros — and names the span after it. The header names the first span of each chain: the sequence, which is the root value's encoding, the schema, and the implementation's private state. An entry in a table page describes one span, and the entries of all live table pages together are the span table.

This is a *piece table*, the structure in which text editors have kept documents under editing since Xerox's Bravo: a document is a list of pieces, each a run of some buffer, and an edit changes the list but never a buffer. Here the buffers are data pages and entries, the pieces are spans, and the list is kept in the file, as the spans' successor links.

A span is known by an id, not by a position, so an edit restates only the spans at the places where it cuts the chain. A span's id stays the same when bytes before it come or go, and the order of the content is recorded nowhere but in the links, so an edit changes what the spans at its cuts show and where they link, and nothing behind them. A splice states one to three entries, counting the span of its new bytes, and a move three to six, wherever it lands and however long the content around it is.

Why links rather than positions. Keying each piece by its position, as an extent tree keys extents by their offset in a file, would make an insertion change the key of every piece behind it, and each of those would have to be written again. Links cost a reader nothing it does not pay anyway: kladde reads the file whole, so it walks each chain once, at open, and keeps a tree over the spans in memory. A tree of spans on file would let a reader find a position without reading what lies before it, which kladde never does, and keeping the lengths in its interior nodes current would restate a path to the root at every edit.

5.2 Logical level

5.2.1 Entries

One kind of entry, in three forms:

form	shows
`Ref(id, length, address, next)`	`length` bytes of the file at `address`, inside the `content` of one `Data` page
`Inline(id, payload, next)`	its payload, 1 to 126 bytes

form	shows
`Zero(id, length, next)`	`length` zeros

`id` is the span's id, and `next` the id of the span after it in its chain, or none at the end of the chain. No span is empty. Writing a span's entry is called *stating* the span, and writing it again *restating* it.

Why `Inline`. Below the encoded size of a `Ref`, a few bytes, the address would be as large as the data, and a short span that a page rewrite moves would cost a data page's bytes as well as an entry. Inline spans ride in the table pages, which the header rewrites every flush anyway; how short a span must be to be kept inline is a writer's choice, below the format's ceiling.

Why `Zero`. Encodings without padding hold few zeros, but `Fixed` values and large preallocated arrays have runs of them, and a `Zero` states any length of them in a few bytes.

5.2.2 Resolution

1. **A span id's entry in the table page of highest epoch is its description.** An epoch holds at most one entry per id, so the winner is unique. Only pages the header reaches through child references are consulted.
2. **Each chain runs from the span its header field names, following `next`, and holds its spans' content in that order.** A header field of `0` names an empty chain.
3. **A file is invalid if a chain names an id that no entry describes, reaches a span twice, or shares a span with another chain.** A reader must check that no span is reached twice, which would otherwise trap the walk in an endless loop at open.
4. **A file is invalid if a reached `Ref` shows bytes outside the `content` of a `Data` page**, or in a page whose epoch exceeds the governing header's (the epoch assertion).

A reader refuses an invalid file rather than return content from it.

Liveness follows: an entry is live if it is the newest of a span that some chain reaches. There are no fragments, no sizes stated apart from content, no tombstones and no counts of mentions, and dropping any other entry can never change what the file holds. A page rewrite states each live span in it again, under the span's own id, and no entry elsewhere changes its meaning.

5.2.3 Rules for writers

- **At most one entry per span id and epoch.**
- **The reach rule: a span that a chain reaches at the end of an epoch, but that no chain reached at the end of the epoch before, is stated in that epoch.**
- **No two reached spans show the same byte of a data page.** This is a writer invariant that readers do not depend on; it keeps a data page's live bytes a plain sum, see design directions.

Why the reach rule. Entries of older epochs stay in their pages until those pages are rewritten, so without it a writer could link in a span id whose newest entry a page rewrite had dropped as dead, and an older entry of the same id, still in a page nobody rewrote, would show its stale bytes. With it, a span id can be handed out again as soon as its span is no longer reached, since the new span's entry is newer than any old one. Readers cannot check the rule, since checking it would need the chains of the epoch before.

5.3 Physical format

Span-table pages follow the general page framing with `kind == Table` (`0x01`), and encode their payload in the `content` field. The header page is always a table page. Every table page contains zero or more references to other table pages followed by zero or more entries. References between table 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 a table page:
table_page := child_ref* 0:byte entry*
```

```
child_ref  := page_number_delta:varint
entry      := id_delta:varint tag:byte body next?
; bit 0 of tag: 1 if the next span is id + 1, and no `next` field follows;
;               0 if a `next` field follows the body
; tag >> 1 = 0:        Ref     body := length:varint address_delta:varint
; tag >> 1 = 1:        Zero    body := length:varint
; tag >> 1 = 2..127:   Inline  body := payload:byte{(tag >> 1) - 1}
next       := varint   ; 0: the chain ends here; else zigzag(next - id)
```

`zigzag` maps a signed integer to an unsigned one, `0, −1, 1, −2, …` to `0, 1, 2, 3, …`, as Protocol Buffers' `sint` types do: `zigzag(v) = (v << 1) ^ (v >> 63)` on 64-bit two's complement.

5.3.1 Delta encoding

A reader decodes a page in order, with a `page_cursor`, an `id_cursor` and an `end_cursor`, all starting at 0.

- For each `child_ref`, `page_cursor` grows by `page_number_delta`, and is then the referenced page's number. Child references end at the delimiter byte `0`, which cannot be a `page_number_delta`: the first child's number is at least 2, since no `child_ref` names a header slot, and each later one exceeds its predecessor, since child references are sorted and the tree names each page once.
- For each entry, `id_cursor` grows by `id_delta`, and is then the entry's id. Entries are sorted by id, so `id_delta` is at least 1; a delta of 0 makes the page invalid.
- A `Ref`'s address is `end_cursor + unzigzag(address_delta)`, after which `end_cursor` is that address plus the `Ref`'s length, where that `Ref` ended. `Zero` and `Inline` entries leave `end_cursor` alone.
- Entries continue to the end of `content`, which `content_size` gives.

`next` never names the span itself, which would be a cycle, so `0` is free to end the chain. A `next` that would name the span itself, id 0, or an id outside the bounds makes the page invalid.

Delta encoding makes a wrong order unrepresentable: entries out of id order would need a negative `id_delta`, which a varint cannot hold. A span written together with its neighbours — by a merge, by evacuation, or when its content was new — gets the next id, and its data follows theirs in the page, so its entry takes about five bytes: one each for the id delta, the tag and the address delta, and two for the length.

5.3.2 A worked example

A drawing whose sequence was one span, `#1`, showing 28 bytes at address 8203 (page 2, right after its framing), is renamed by a splice of 9 bytes over its first 3. Afterwards, `#2` holds the 9 new bytes inline and links to `#1`, and `#1` is restated to show the last 25 bytes, at address 8206; the header's `sequence_head` becomes 2. If both entries go in the header, its `content` is:

```
00                                  no child references
01 00 19 9C 80 01 00                #1: id 0 + 1; Ref, next follows; length 25;
                                    address 0 + unzigzag(16412) = 8206; next: end of chain
01 14 57 6F 72 6C 64 20 6D 61 70 01 #2: id 1 + 1; Inline of 20 >> 1 − 1 = 9 bytes, next follows;
                                        "World map"; next = 2 + unzigzag(1) = 1
```

Had the splice landed in the middle of a sequence of a million spans, the same two entries, and one more for the span its start cut in two, would have described it: every span behind it keeps its entry although its bytes moved.

5.3.3 Notes on the physical format

- The root table page *is* the current header slot, so before its `content` it carries the file-header fields, all covered by the header CRC.
- A `child_ref` stores only a 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.
- Nothing in the format constrains *which* page an entry lives in or how the tree is shaped; those are writer policies. The tree is a *directory*, not a search structure: nothing descends it by key.

- A reader should verify that the `child_refs` form a **tree**, so that cyclic references, or a page named twice in a graph that is not a tree, cannot trap it in a loop or have it decode a page twice.

5.4 Bounds

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

- **Span ids** are 32-bit, and range from 1 to `2^32 - 2`: 0 means "none" in the header and in `next`, and the largest id is left out so that `id + 1`, which an implicit successor names, is always a valid id. So a file holds at most `2^32 - 2` spans per moment, and a varint-encoded id takes at most 5 bytes.
- **Page sizes** are uniform throughout a file and given by the header's `log2_page_size`, currently 4 KiB with 8, 16, 32, and 64 KiB reserved, so an offset within a page fits in 16 bits.
- **A `Ref`'s length** is at most `MAX_PAGE_CONTENT`, the page size minus the 15 bytes of page framing: at most `2^12 - 15` bytes with 4 KiB pages, and `2^16 - 15` with 64 KiB ones; its varint takes at most 3 bytes.
- **An `Inline` payload** is 1 to 126 bytes, and **a `Zero`'s length** 1 to `2^32 - 1`.
- **Page numbers** are 32-bit, so **addresses** fit in 48 bits (49 to point at the end of the file), an `address_delta` takes at most 7 bytes, and the largest **file** is `2^(32+12)` bytes, about 17.6 TB, with 4 KiB pages.
- **Each chain's content** is shorter than `2^48` bytes, so a position in the sequence takes at most 7 bytes as a varint.
- **An entry's framing** — its encoded length without an `Inline` payload — is at most **21 bytes**: a `Ref` with 5 bytes of `id_delta`, a tag byte, 3 bytes of length, 7 of `address_delta` and 5 of `next`.
- **The live `Ref`s showing bytes of one `Data` page** have lengths summing 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; the bound binds only files that let several reached spans show the same byte, which is legal and which a reader must therefore survive.
- **An encoded transaction**, without framing, takes at most `2^32 - 1` bytes, so that its length prefix fits in 32 bits.
- **An external reference** names 1 to `MAX_PAGE_CONTENT` bytes inside one page's `content`, as a `Ref` does, so its length takes at most 3 bytes and its address at most 7. Trying to encode a larger transaction must fail rather than silently cut it 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 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. **Copy-on-write clones are not pursued**: they add complexity and open a zip-bomb-like attack vector, and duplicating a value writes its encoding instead.

Per-page epochs versus per-entry epochs. Per page is cheaper and imposes exactly one discipline — a rewrite states the newest entries — which consolidation requires anyway.

Overwrites inside a span: split or rewrite, not shadow. An overwrite of bytes inside a span either splits the span in three around the written bytes, or writes the whole span again with the new bytes; which one is a writer's choice (the flush). A third form, an entry that overwrites part of a span by offset, would cost one entry and write no other byte, but it would bring back per-byte resolution, fragments inside spans, and liveness that counts them: most of what one entry per span removes. It is the option to revisit if overwrites inside long, cold spans turn out to dominate a workload. A smaller first step would be entries that only shrink or relink a span stated by an `Inline` entry of an earlier epoch, sparing a restatement of its payload of up to 126 bytes. They would be restricted to spans that a chain reached at the end of the epoch before, by the reach rule; they would keep the `Inline` entry alive with a dead part of its payload; "newest wins" would then apply separately to an inline span's payload, size and successor, so a span could have two live entries; and a page rewrite that writes the `Inline` entry again would have to state the span's current size and successor with it. Neither is adopted until measurements motivate it.

The `Inline` threshold. The encoding fixes the ceiling at 126 bytes; the policy threshold belongs below it, and is an implementation choice rather than a format one. See the implementation notes.

Chapter 6

The sequence

What a kladde file's content is: one byte sequence holding the root value's encoding, beside the schema and the implementation's private state, each a chain of spans.

6.1 Three chains

The header names three chains, and each holds a byte string:

chain	header field	holds	changed by
the **sequence**	`sequence_head`	the root value's encoding, by the root type's descriptor	the journal, and flushes
the **schema**	`schema_head`	the descriptor table of the root type, root first	flushes only
the **private chain**	`private_head`	whatever the writing implementation keeps from one session to the next (rules)	flushes only

The journal edits only the sequence, so no journal record needs to say which chain it edits, and every position in the journal is a position in the sequence, counted in bytes from the root value's first byte. The schema changes only when a flush writes it: when a file is created, or upgraded to a new schema before the application gets it. A file whose root fingerprint matches the reader's never needs to decode its schema chain (the fast path).

Why chains of their own rather than a prefix. The schema could be a prefix of the sequence, which would work as well; but it would offset every position by the schema's length, and a reader would decode the schema to find where the root begins. Chains cost nothing extra in the span format.

6.2 One sequence, holding the root value's encoding

The sequence is the encoding of the root value, with every value inline in its parent; there is no allocation and no pointer. A container states how many elements it holds, text ends in a byte that UTF-8 never contains, no value states its own encoded size, and every value decodes front to back from its type's descriptor.

For example, with these types,

```
struct Drawing { title: Text, shapes: Sequence<Shape> }
enum Shape { Circle { r: f32 }, Path { points: Sequence<(f32, f32)> } }
```

a drawing titled "Map" with a circle of radius 1 and a path from (0, 0) to (1, 1) is this sequence of 28 bytes:

```
4D 61 70 FF                 title: "Map", and the end of the text
02                          shapes: 2 elements
00 00 00 80 3F              Circle (variant 0), r = 1.0
01 02                       Path (variant 1), 2 points
00 00 00 00 00 00 00 00       (0.0, 0.0)
00 00 80 3F 00 00 80 3F       (1.0, 1.0)
```

Nothing in it is a pointer, a size of an allocation, or a byte of padding. Renaming the drawing to "World map" is one journal record, `Splice(0, 3, "World map")`, and every byte behind it moves by 6 without being mentioned; on file it is two span entries, wherever in the sequence the splice lands.

Why one sequence.

- **The description stops depending on the data model.** When every value lives in an allocation of its own, every value costs a pointer in its parent, a description naming its allocation, and a size, so the description grows with the number of values, and short strings and short lists dominate that number. With one sequence, the description depends only on how a writer cuts the sequence into spans, which it chooses and the data model never sees.
- **The unit of description and the unit of the data model stop pulling against each other.** A cheap description wants few allocations, so values move inline into long ones; a cheap splice wants short allocations, since content keyed by its offset in an allocation makes a splice restate everything behind it. Separating the two removes the trade-off rather than choosing a point on it.
- **Everything that exists because of allocations goes with them**: ownership, freeing, publishing a pointer before freeing what it replaced, tombstones, id recycling, and leaks. Removing a value is one splice, whatever it held, and nothing can leak, since nothing can be unreachable.
- **The file is its own serialization.** A tool decodes the whole content from the schema, with no pointer graph to follow and no opaque container to stop at.

It gives up reaching a value on file without decoding what lies before it, which kladde never does, since it reads in bulk. And positions change whenever something before them changes size, so an implementation tracks them in memory: each container knows where its children start, relative to itself (the reference algorithms). Allocations and the address table records the design this replaced.

6.3 What a valid sequence is

At every transaction boundary, the sequence is the canonical encoding of exactly one value of the root type: it decodes by the root descriptor to the end, with no byte left over, and every encoding in it is canonical — minimal varints, valid UTF-8, discriminants that name variants, integers in range of their type, and zeros wherever a `Fixed` value pads. A reader refuses a file whose sequence, after the journal is replayed, is not.

Inside a transaction, anything goes: the states between its records are never observed. This is the journal's one ordering rule, and it is all that is left of the ordering discipline allocations needed: nothing can dangle and nothing can leak, so nothing has to be published before something else is freed.

A canonical encoding makes the sequence a function of the root value, so two implementations that hold the same value hold the same bytes, and a test can compare a file's sequence with the canonical encoding of the value in memory, byte for byte.

6.4 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; bytes within a data page are live if a span a chain reaches shows them. 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 — is its own business, and nothing about it is persisted.

6.5 Reclamation is not part of this specification

Whether an implementation reclaims garbage, when it does so, and which pages it chooses are unconstrained. A consolidation step writes live content into fresh pages and states the spans that show it again, producing a file indistinguishable from one that happened to be laid out that way from the start. Two implementations with 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 the content of any chain. How the content is cut into spans, and which span ids it uses, are not content: a writer may split, merge, renumber and restate spans freely.

6.6 Partial loading, later

Loading part of a file means skipping a subtree without decoding it and without reading its spans, and one sequence offers neither. A segment would therefore be a chain of its own, named from the sequence by a reference: one coarse kind of pointer, reintroduced deliberately, for exactly the subtrees an application declares. The format already has three chains, so a fourth kind changes the header's fields and a descriptor kind, not the span machinery; whether references to segments may move, and how their transactions relate to the sequence's, is open.

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 the sequence. 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.

Each epoch has a journal of its own. The journal of epoch `E` collects the transactions issued between flush `E - 1` and flush `E`, in a chain of journal pages whose first page the header of epoch `E - 1` names. Flush `E` folds it into table and data pages, and the header it commits names the first page of the journal of epoch `E + 1`.

A *record* is what the journal holds; an *operation* is what an application's call does, which may take several records, as a push onto a vector takes two, the element and the count.

The journal is deliberately **type-agnostic**. It records effects on bytes of the sequence, not application-level operations: there is no "push element", no "insert key", only overwrites, splices, and moves of bytes.

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 hundred edits that cancel out still record a hundred edits' worth of byte changes, because nothing in the log knows they were edits.

7.2 Record kinds

Four records, on positions in the sequence:

record	effect
`Write(position, bytes)`	overwrites `[position, position + bytes.len())`
`Splice(position, removed, bytes)`	replaces the `removed` bytes at `position` with `bytes`, shifting what follows
`Move(from, length, to)`	cuts `[from, from + length)` and inserts it at `to`, a position in the sequence without the cut bytes

record	effect
`Paste(position, removed, cut)`	replaces the `removed` bytes at `position` with the bytes of a cut that an earlier record of the same journal made

Records take effect in the order they were recorded, and a record's positions refer to the sequence as the records before it left it. To move `bc` between `d` and `e` in `abcde`, giving `adbce`, `from` is 1 and `length` 2, which leaves `ade`, and `to` is 2, between `d` and `e` there: `Move(1, 2, 2)`.

A record whose range does not lie inside the sequence makes the file invalid, as a transaction whose checksum matches but whose records do not decode does: it is a writer's bug, never a torn write, and a reader refuses the file. With one sequence, there is nothing for a record to bring into existence, so no record needs a meaning outside it.

A move of `B` behind `C` in `A B C D` is also a move of `C` in front of `B`, and which one a writer records changes nothing on file: the resulting sequence is the same, cut at the same three places. A container records the move of the element it was asked to move.

There is no record to allocate, free, resize or copy. Duplicating a value writes its encoding: the flush would have to write a copy's bytes again anyway, to keep every data byte shown by one span, so a copy record would save journal bytes only.

7.2.1 Cuts

Every record that removes bytes — a `Write`, and a `Splice` that removes some — makes a *cut* of them, which a later `Paste` of the same journal may insert elsewhere, once, and which is gone when the journal is folded.

- A `Paste` names its cut by how many cuts of the journal were made after it: 0 for the latest. With `removed` equal to the cut's length it overwrites, and with 0 it inserts.
- A cut is pasted at most once, so that no data byte is shown twice. A reader refuses a `Paste` that names a cut already pasted, or one its journal did not make.
- A cut that nothing pastes is removed bytes, as after any splice: nothing frees it, and nothing can leak.
- A `Write` is a `Splice` that removes as many bytes as it inserts, a byte shorter in the journal, so a writer records a `Write` wherever the lengths agree; that its overwritten bytes make a cut too keeps that choice from deciding whether the old value can be pasted elsewhere.
- A `Write` of no bytes makes no cut, and neither does a `Splice` that removes nothing.
- A `Move` makes no cut either: the bytes it cuts it inserts again at once, so nothing is left for a `Paste` to name, and counting it would only make a `Paste`'s distance depend on the moves between it and its cut.
- A cut and its paste may lie in different transactions, as long as both lie in one journal; recovery replays a journal from its start, so every cut that a recovered `Paste` names is among the transactions it replays.
- `Move(from, length, to)` is `Splice(from, length, [])` followed by `Paste(to, 0, 0)`, and stays as the shorter way to write it.

Why cuts. Every value is its bytes, so moving a value from one container to another would otherwise mean writing its encoding again, and the journal and the flush would carry all of its bytes. A cut and its paste carry none: the flush resolves them as a move, which states three to six spans however large the value is. Naming cuts by distance rather than popping them off a stack lets a writer paste its cuts in any order, so two values that swap containers need no care about which is pasted first.

Why as long as the journal, and not longer. Within one journal every cut is still at hand: its bytes lie in the committed sequence, in the journal itself, or in its payload pages, which the fold and recovery all read, so a cut can outlive its transaction at no cost to the file. That serves an application that removes a value in one transaction and inserts it again in a later one — undoing a deletion is the common case — which would otherwise write the value's encoding again. A cut that outlived its journal would need its bytes kept reachable from the header, as content of their own, after the flush that folds their removal; deleted

content would then stay in the file, which is not what an application deleting it expects. So a cut is a *weak* promise: an implementation pastes it while its journal lasts, and encodes the value again otherwise. The price is that a reader keeps the length of every cut of the journal, to check the positions after it, which the journal's length bounds, and an implementation keeps what it needs to resolve the journal's cuts until the journal is folded.

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.

7.4 Ordering

> **Every transaction takes a sequence that decodes to one that decodes.**

Replay stops only at transaction boundaries, so the states inside a transaction are never observed and may be anything, while the state at every boundary is one that recovery can land on, and must be a valid sequence. A container whose edit changes two places — the element and its count, for a push — makes both in one transaction.

That is the whole discipline. Nothing in the sequence points at anything else, so nothing can dangle, and nothing can become unreachable, so nothing can leak. The conformance suite checks it mechanically: truncate the journal at every byte, replay, and decode.

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

Each position is written as its distance from where the record before it left off, and the record's kind rides in the low two bits of the same varint:

```
record  := head:varint (write | splice | move | paste)
; x := unzigzag(head), a signed integer
; kind := x & 3, and delta := x >> 2, shifting arithmetically
write   := bytes                          ; kind 0; position := cursor + delta
splice  := removed:varint bytes           ; kind 1; position := cursor + delta
move    := length:varint to_delta:varint  ; kind 2; from := cursor + delta,
                                          ;   to := from + unzigzag(to_delta)
paste   := removed:varint cut:varint      ; kind 3; position := cursor + delta
bytes   := length:varint byte{length}
```

So a writer computes `head = zigzag((delta << 2) | kind)`, with `zigzag` as in the span table.

The cursor is where the record before left off: the end of what a `Write`, `Splice` or `Paste` put in place — `position` plus the bytes it inserted — and the end of the moved run at its destination, `to + length`, for a

`Move`. It starts at 0 at the start of each epoch's journal, which is always replayed from its start, and carries over from one transaction to the next.

A `Write` of no bytes is the escape to further kinds of record, since it changes nothing: its head is followed by `escape:varint`, which says what record follows.

```
escaped  := 0x00 0x00 escape:varint (seal | external)
                                          ; the head of a Write with delta 0,
                                          ;   and its length 0
seal     := page:varint                   ; escape 0
external := record                        ; escape 1: a Write or Splice
                                          ;   whose bytes are external
```

An escaped record must have delta 0, and leaves the cursor where it was.

Two escapes are defined: 0, the seal, and 1, a record whose bytes are external; a reader refuses any other until a revision defines it.

Every position, `removed` count, `length` and `to` is below `2^48` (bounds), and a record's range must lie inside the sequence. A `Paste`'s `cut` must name a cut of its own journal that is not pasted yet; its length is the cut's.

Why relative positions. Edits mostly land near the one before, so a delta from −16 to 15 fits in the head's one byte with the kind, where an absolute position takes three bytes in a sequence of 50 KB and four in one of 100 MB, besides a byte for the kind. A jump across most of the sequence costs at most one byte more than an absolute position would, since zigzag adds one bit. A keystroke is then a `Splice` of four bytes — the head, nothing removed, a length of one, and the character — and a transaction of nine with its length and checksum. A `Move`'s `to` counts from its `from`, since restacking moves bytes a short way.

7.5.2 Seals

A seal ends its epoch's journal and names the page where the journal of the next epoch begins, so that a flush can fold one journal while the application appends to the next.

- A seal is a transaction of its own, holding the seal record and nothing else, and nothing follows it in its journal.
- Its `page` is the first page of the next epoch's journal, chosen as any other journal page is: a reusable page, or a new page at the end of the file, never a page of the journal it ends.
- From the moment the seal is appended, the next journal belongs to the world of every header whose journal leads to the seal, and the reuse rule protects its pages as it protects the journal itself.
- The next journal is an ordinary journal of epoch `E + 1`: its CRC chain is salted with `E + 1`, its cursor starts at 0, and its cuts are its own. The header that the flush folding journal `E` writes names the same page as its `journal_pointer`.

A writer need not seal: a flush that folds its journal without one is as valid as ever, and the header it writes names the next journal, as before.

Why a seal. Without one, the next journal exists only once the flush's header names it, so a transaction appended while a flush is in progress would belong to a journal that recovery cannot reach if the application crashes before that header is written, although the call that recorded it had returned. With a seal, recovery reaches the next journal through the journal it folds, whichever of the two headers governs. A writer can then let the application append while it folds, writes and syncs, on the caller's thread or another; whether it does is its own choice, and the format is the same either way. The seal reuses the escape that a `Write` of no bytes was reserved for, so it costs no kind of its own.

7.5.3 External bytes

A `Write` or a `Splice` may keep its bytes *external*: in a framed `Data` page of their own, a *payload page*, which the record names by length, address and checksum, and which the flush that folds the journal takes into the file as a data page without writing its bytes again. Bytes in the journal are written twice, once by the append and once by the flush that writes them into data pages, and the journal pages they took are holes once the journal is folded; external bytes are written once, and their pages stay in use.

```
external := record              ; a Write or Splice, after escape 1
; in it:
bytes    := length:varint address:varint crc:byte{4}
```

- `length` is 1 to `MAX_PAGE_CONTENT`, and `[address, address + length)` lies inside the `content` of one page, as the bytes a `Ref` shows do; `address` is absolute, a byte offset in the file, and `crc` is the CRC-32C of those bytes.
- The record after escape 1 is a `Write` or a `Splice`; a `Move`, a `Paste`, or another escape there makes the file invalid. It moves the cursor and makes a cut as it would with its bytes inline.
- A payload page is a `Data` page whose `epoch` is that of the journal whose transactions name it. It is written before the first transaction that names it is appended, to a page that is reusable then or added at the end of the file, and it belongs to the world of every header whose journal leads to that transaction from the moment the transaction is appended.
- While its journal is the one being appended to, a payload page may be written again to append bytes after its content, as the journal's last page is; it is written whole each time, framing included, and changes no byte a transaction already names.
- No two references of one journal name overlapping bytes, a writer invariant that readers need not check, as for spans.
- A flush that folds a payload page takes it into the header it commits only if the page's epoch is not above its own, by the epoch assertion: a payload page of a later journal, reached through a seal by a flush that folds both, has its bytes written again.

Why an escape, rather than a flag in the head or in the length. A flag in the head would halve the deltas that fit its one byte, from −16 to 15 to −8 to 7, for every record, and would mean nothing for a `Move` or a `Paste`; a flag in the length of `bytes` would cut the inline payloads whose length fits one byte from 127 bytes to 63. The escape costs three bytes per external record and nothing for any other: 0.07 % of a reference to a full page, and 0.15 % of one to 2 KB.

Why an absolute address. A distance from where the journal's previous reference ended would mostly be the 15 bytes of framing from one full page's content to the next, a byte against four in a file of 2 MB to 256 MB; the three bytes it would save are as small a share as the escape's, and an absolute address needs no cursor carried across records and transactions.

Why a checksum per reference. The checksum of a transaction covers its records as they are in the stream, so recovery validates a transaction before it decodes it; feeding the external bytes into the CRC chain instead would make recovery decode unvalidated records, to find which pages to read, and would tie the chain's definition to the record encoding. And the payload page's own framing cannot stand in for the checksum: when a crash loses header `E` after a session wrote payload pages for journal `E + 1`, the next session issues epoch `E` again, and a page it takes for journal `E + 1` may still hold the lost session's payload page, framed and of epoch `E + 1`; if a power cut then kept a transaction but not the new write of its page, the stale framing would validate. No zeroing can help there, as it helps at the start of a session, since the stale pages could be any of the reusable ones. The checksum costs four bytes per reference, 0.1 % of a reference to a full page and `4 / r` of one to `r` bytes, and needs no salt: a page that holds exactly the bytes a reference names holds the right bytes, however it came to hold them.

Why a payload page's framing is checked all the same. A torn write can leave a page's new content under its old framing, and the flush takes the page into a header's world, where every page's framing must validate; requiring the journal's epoch also keeps a reference out of the governing header's world, whose pages carry that header's epoch at most.

Why appends to a payload page are safe. A tear needs a power cut, and under a power cut the guarantee covers only the transactions a completed flush folded, whose payload pages are data pages of a committed header that nothing writes again. A torn append damages the bytes earlier transactions of the same journal name, or the page's framing, so their checks fail and recovery stops before the first of them, as a torn append to the journal's last page does; it reaches back further, to the oldest transaction whose bytes share the page, and that is bounded by the epoch either way.

7.5.4 Framing

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

`length` counts the bytes of the records, and `crc` is the value of the journal'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, break a bound, reach outside the sequence, or paste a cut they may not, 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.

7.5.5 Pages

An epoch's journal is a singly linked chain of journal 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 journal'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 journal 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 journal.

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.

7.5.6 The CRC chain

Every checksum in an epoch's journal is a value of one running CRC-32C over the journal's chain input, using the CRC-32C specified for the file. The chain input is the journal's epoch as 8 little-endian bytes — one more than the epoch of the header that names the journal — 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 journal implies it. This is what lets a flush hand the journal of epoch `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 journal 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 journal 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. At a complete transaction that is a seal, continue the walk at the page it names, as the journal of the next epoch: the chain restarts, salted with that epoch, and so do the cursor and the cuts.
4. At a complete transaction that names external bytes, read each reference's page: if the page lies past the end of the file, its framing fails, it is no `Data` page of the journal's epoch, the range does not lie inside its content, or the bytes there do not match the reference's `crc`, the page's write did not reach the disk, and the transaction belongs to the torn tail. A reference whose `length` is 0 or above `MAX_PAGE_CONTENT`, or whose range would cross a page boundary, is a writer's bug, since a torn write cannot produce a matching transaction `crc`, and makes the file invalid.
5. Stop at the first check that fails, or at a `journal_pointer`, `next` or seal that points past the end of the file. The complete transactions before that point are the journal; whatever follows them is the torn tail.
6. 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 the journal of epoch `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 it names, for which zeros suffice, unless that page lies past the end of the file; a flush that seals writes it, and makes the write durable with an `fsync`, before it appends the seal. The `fsync` is what the seal needs and the header does not: a header is written only after an `fsync` that covers the zeros, but a seal is appended with nothing in between, so a power cut could keep the seal and lose the zeros, and the lost journal on the page would validate after the seal. The epoch salt tells every journal apart except in one situation. When a crash loses header `E` after transactions were appended to the journal 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 journal'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 journal 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 is started. The protocol that commits this is in Durability.

A flush folds its whole journal, 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 — dropping a write that a later write fully supersedes, resolving a cut and its paste as a move,

stating bytes a splice shifted without writing them again, stating external bytes where their payload page holds them. It may reorder freely, so long as the resulting sequence is the one an in-order replay produces: every byte of the sequence has one right answer, so a fold's freedom is to reach that answer by any route, and to cut it into spans however it likes.

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: a field is decoded 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 *encoding*, not merely identity, carries enough information to actually bridge a difference rather than just report one. A reader whose struct gained a field can decode the writer's values by the writer's descriptors, and read the fields it does share 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, what each carries, and how its values are encoded in the sequence
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 encoding for this field match what my decoder 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 every value is **encoded**, and how descriptors are **modeled, serialized, and fingerprinted**.

The descriptor table sits in the file's schema chain, and the root fingerprint in the header. 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 encodes into the sequence, and what its bytes mean.

Status: settled and implemented. This document is normative.

9.1 Terminology

- **Type descriptor** (*descriptor*): a node describing one value type's encoding.
- **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, within the rules of nesting.

9.2 The kinds

A descriptor is one of nine kinds: **Primitive**, **Struct**, **Enum**, **Array**, **Sequence**, **Map**, **Text**, **Opaque**, and **Fixed**. Every type has **one encoding**, which a reader decodes front to back from the type's descriptor, below.

9.2.1 Primitive

A scalar, identified by a **primitive code**.

code	type	encoding	fixed form
0	`u8`	one byte	the same byte
1	`u16`	unsigned LEB128	2 bytes, little-endian
2	`u32`	unsigned LEB128	4 bytes, little-endian
3	`u64`	unsigned LEB128	8 bytes, little-endian
4	`i8`	one byte, two's complement	the same byte
5	`i16`	zigzag, then LEB128	2 bytes, little-endian two's complement
6	`i32`	zigzag, then LEB128	4 bytes, little-endian two's complement
7	`i64`	zigzag, then LEB128	8 bytes, little-endian two's complement
8	`f32`	4 bytes, little-endian IEEE-754	the same bytes
9	`f64`	8 bytes, little-endian IEEE-754	the same bytes
10	`bool`	one byte, `0x00` false, `0x01` true	the same byte
11	`char`	UTF-8, 1 to 4 bytes	the Unicode scalar value, 4 bytes, little-endian

Zigzag maps `0, −1, 1, −2, …` to `0, 1, 2, 3, …`, so that a number of small magnitude takes one byte whatever its sign. The *fixed form* is what the type's values take inside a `Fixed` 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.

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, and must refuse a table that uses one it does not implement.

code	type
12	`u128`
13	`i128`
14	`f16`, IEEE-754 binary16
15	`bf16`, bfloat16
16–19	`u24`, `u40`, `u48`, `u56`
20–23	`i24`, `i40`, `i48`, `i56`

9.2.2 Struct

An ordered, fixed set of named fields, encoded back to back — 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 order of the encodings. 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 value that the variant it selects holds, its **payload**.

Carries:

- `name`: UTF-8, diagnostics only.
- `discriminant_width`: the bytes the discriminant takes in the **fixed form** — 1, 2, 4, or 8; every discriminant value must fit in it. The encoding itself writes the discriminant as a varint.
- `variants`: each `(discriminant_value, variant_name, payload)`, canonically ordered by ascending `discriminant_value`; discriminant values are distinct, and `payload` is a reference.

Each discriminant value, each variant name, and each variant's payload type is significant. Variant *declaration* order is **not**: a value's encoding 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 encoding nor the fingerprint.

Every variant holds exactly one value. A variant that holds nothing holds the empty Struct, which encodes to no bytes; a variant that holds several fields holds a Struct of them; and a variant that holds one value of any type holds that type directly. So in Rust, a unit variant `Nil` holds the empty Struct, a tuple variant `Element(Element)` or `Text(String)` holds its field's type, a struct variant `Point { x: f32, y: f32 }` holds a Struct of its fields named after the variant, and a tuple variant of several fields `Pair(u8, u8)` a Struct of positional fields.

Why one value, always. Languages disagree about what a variant holds: Rust has unit, tuple and struct variants, while a C++ `std::variant`, a C++ class hierarchy, Swift's enums with associated values and TypeScript's discriminated unions all hold exactly one value per alternative. Describing a variant by one payload lets every one of them map to the same descriptor with no choice left to the programmer: Rust's `Element(Element)` and a struct variant with `Element`'s fields describe the same bytes and get the same

fingerprint, as a struct's own name already does not matter. And holding *always* one value, rather than at most one, keeps every variant the same shape: a valueless variant, a C++ alternative that is an empty struct and Rust's `()` all hold the empty Struct, and need no rule to say they agree.

An `Option<T>` is an enum, `None` with discriminant 0 holding the empty Struct, and `Some` with discriminant 1 holding a `T`.

9.2.4 Array

`len` values of one element type, back to back, with no count.

Carries `element` (a reference) and `len`. An Array is deliberately *not* interchangeable with a Struct of `len` identically-typed positional fields: the encodings coincide, but the kind tags differ and therefore so do the fingerprints. The kind exists to avoid spelling out a large `len` as that many fields.

9.2.5 Sequence

Any number of values of one element type: their count, then the values.

Carries `element` (a reference). A vector is a Sequence; a map is a Map.

9.2.6 Map

Any number of entries, each a key and a value: their count, then each entry's key and value.

Carries `key` and `value` (references). The entries are in the order the writing library keeps them; the format neither sorts them nor checks that keys are distinct, which is the library's business, as is how it finds a key in memory.

Why a kind of its own. A Map encodes exactly as a Sequence of two-field Structs would, but every language has a native map that a reader can decode one into, and a tool can render one as a JSON object rather than as an array of pairs. It also leaves nothing to agree on: a Sequence of Structs would need every implementation to name the two fields alike, or the same map would get two fingerprints.

9.2.7 Text

UTF-8 text, ended by the byte `0xFF`. Carries nothing.

Why a terminator rather than a count. UTF-8 never contains `0xFF`, so no byte of text needs escaping, and an edit of text is then one splice, where a count would add a second record — and a second place where the chain of spans is cut — to every keystroke. A decoder finds the end by scanning for the terminator, which it does while it checks the UTF-8 anyway; text below 128 bytes takes the byte a count would have taken, and longer text saves the count's further bytes. Only text can have a terminator, since a terminator must be a byte the content cannot hold, so bytes and sequences of any other type keep their count.

9.2.8 Opaque

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

Used for types with hand-written encodings that no structural descriptor fits: externally serialized payloads, say.

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.
- `fixed_size`: the number of bytes every value takes, or none, in which case each value is prefixed with its length.
- `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 encoding-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 encoding changes. A structural descriptor needs no such trust, which is a standing argument for preferring structural descriptors wherever a type admits one, and reserving Opaque for encodings that genuinely are not decomposable. Whether a fixed size or a length prefix per type is enough, or whether some opaque types want both modes, is open.

9.2.9 Fixed

The values of a wrapped type, each encoded at the same size: its **fixed form**.

Carries `target` (a reference). `Fixed(T)` is a type of its own, with the values of `T`: it lays out everything inside `T` at its largest, so every field lies at a constant offset whatever the value, and a container of them finds its `i`th element by multiplication. It applies to all of `T` and not to what wraps it: `Fixed(Option<u32>)` takes five bytes for every value, while `Option<Fixed<u32>>` takes one for `None` and five for `Some`.

9.3 Encodings

kind	encoding
Primitive	as its code gives
Struct	its fields' encodings, concatenated, in declaration order
Enum	the discriminant as unsigned LEB128, then the selected variant's payload's encoding
Array	`len` element encodings
Sequence	the number of elements as unsigned LEB128, then the element encodings
Map	the number of entries as unsigned LEB128, then each entry's key's encoding and its value's
Text	the UTF-8 bytes, then `0xFF`
Opaque	`fixed_size` bytes, or, without a fixed size, the number of bytes as unsigned LEB128 and then the bytes
Fixed	the target's fixed form, below

The fixed form of a type is:

kind	fixed form
Primitive	the fixed form of its code
Struct	its fields' fixed forms, concatenated
Enum	the discriminant in `discriminant_width` bytes, little-endian, then the variant's payload's fixed form, then zeros up to the size of the largest payload's
Array	`len` element fixed forms
Opaque	its encoding, if it has a fixed size
Fixed	the target's fixed form

Sequence, Map, Text, and an Opaque without a fixed size have **no** fixed form, and neither has any type that holds one of them, or itself.

The canonical encoding is the only one a writer may produce, and a reader refuses any other: varints are minimal, text is valid UTF-8, discriminants name variants, integers lie in range of their type — a `u16`'s varint at most `2^16 - 1`, an `i16`'s zigzag likewise — `bool`s are 0 or 1, a `char`'s fixed form is a Unicode scalar value, and an enum's padding in the fixed form is zeros. So a value has exactly one encoding, and the sequence is a function of the root value.

9.3.1 A worked example

With `struct Point { x: f32, label: Text, n: u32 }`, the point `(1.5, "ab", 300)` encodes as

```
00 00 C0 3F      x: 1.5
61 62 FF         label: "ab", and its terminator
AC 02            n: 300 as LEB128
```

and `Fixed(Option<u32>)` holding `Some(300)` as `01 2C 01 00 00`: the discriminant in one byte, then 300 in four, little-endian; `None` is `00 00 00 00 00`.

9.4 Fixed-size types

A type whose every value encodes to the same number of bytes is fixed-size, and the descriptors say which types are, so a reader computes it as a writer does:

- `u8`, `i8`, `bool`, `f32` and `f64` are, with sizes 1, 1, 1, 4 and 8; the other primitives are varints or UTF-8, and are not;
- a Struct is if all its fields are, with the sum of their sizes;
- an Enum is if every discriminant is below 128, so that its varint takes one byte, and every variant's payload is fixed-size with the same size: its size is 1 plus that size;
- an Array is if its element is, with `len` times the element's size;
- an Opaque is if it declares a fixed size;
- a `Fixed` always is, with the size of its target's fixed form;
- Sequence, Map and Text never are, nor is a type that holds itself.

A fixed-size type's fixed form is its encoding byte for byte, except where an enum's `discriminant_width` exceeds the one byte its encoding spends on the discriminant: a writer may set a wider width so that the fixed form keeps its size when variants are added.

9.5 Rules of nesting

A table must keep two rules, which a reader checks when it decodes one:

- **A `Fixed` holds a type that has a fixed form.**
- **A type reaches itself only through an Enum, a Sequence or a Map**, whose discriminant or count can end the recursion, and never through Structs, Arrays and `Fixed` alone, whose values would be infinite. A tree whose nodes hold their children inline is a Struct holding a Sequence of itself; a list is an Enum one of whose variants holds itself. An Opaque's parameters are not containment: an Opaque says nothing of how its value holds them, so a type may reach itself through one. The rule asks only that recursion *can* end; an Enum whose every variant holds itself has no finite value, and is legal but useless.

A decoder of a recursive type nests as deeply as the file says, so it limits its depth, or does not recurse, and refuses a file beyond its limit rather than overflow its stack.

9.6 References and the root

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

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

Why the model is enough for evolution. Every encoding is decodable from the descriptors alone, so a reader can skip, or read, a writer's values by the **writer's** graph, even when its own types 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	`string(name) varint(field_count)`, then per field `string(field_name) reference(type)`
129	Enum	`string(name) byte(discriminant_width) varint(variant_count)`, then per variant `varint(discriminant_value) string(variant_name) reference(payload)`
130	Opaque	`string(library_name) string(type_name) varint(major) varint(minor) varint(patch) varint(size_mode) varint(param_count)`, then per parameter `reference(type)`
131	Array	`reference(element) varint(len)`
132	Map	`reference(key) reference(value)`
133	Sequence	`reference(element)`
134	Text	nothing
135	Fixed	`reference(target)`

An Opaque's `size_mode` is `0` for a value prefixed with its length, and one more than the size for a type with a fixed size: `n + 1` for `n` bytes. Tags `136..=255` are unassigned and reserved for future kinds, and a reader refuses a table that uses one.

A decoded table must also keep the rules of nesting, and every enum's discriminant values must fit its `discriminant_width`; a reader refuses one that does not.

A reader refuses every table that is not the canonical encoding of the table it decodes to: a varint longer than its value needs, enum variants out of ascending discriminant order, two variants with one discriminant, or bytes after the last descriptor. So a table that decodes re-encodes to its own bytes, and two tables that decode alike are byte-identical, which is what lets fingerprints and the file's schema chain be compared as bytes.

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 keep their given order, being positional type arguments; a Map emits its key before its value, and an Array, a Sequence and a `Fixed` have one reference each.

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, and the content of a file's schema chain. A table holds at least one descriptor, every reference lies inside it, and no byte follows its last descriptor.

A table of a vector of text, `Sequence(Text)`, is the four bytes `02 85 01 86`: two descriptors, a Sequence (133) of descriptor 1, and Text (134).

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 has universal test vectors, it is in the standard library of most languages and a few hundred lines of code in those without one, such as C and C++, 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;
- an Array's length, and an Opaque's library name, type name and size mode.

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 The computation, exactly

Every descriptor starts white. `visit(node, depth)` colours `node` grey, records its `depth`, feeds a fresh SHA-256 the node's **hash input**, colours it black, remembers the first 16 bytes of the digest as its fingerprint, and returns them. The file's root fingerprint is `visit(0, 0)`.

A node's hash input is its canonical encoding, with three changes:

- a Struct's and an Enum's `string(name)` is left out;
- an Opaque's `varint(major) varint(minor) varint(patch)` becomes `byte(stable) varint(leading)`, with `stable` 1 if `major > 0` and 0 otherwise, and `leading` the major version if `major > 0` and the minor version otherwise;
- every `reference(child)` becomes a **token**, emitted in the order the encoding emits references, variants in ascending discriminant order:
  - if `child` is white, `byte(0)` followed by the 16 bytes of `visit(child, depth + 1)`;
  - if `child` is black, `byte(0)` followed by its remembered 16 bytes;
  - if `child` is grey, which makes the edge a back edge, `byte(1)` followed by `varint(depth - depth[child])`.

So the token for a child is computed in the middle of its parent's input, which is why the hash state of every node on the stack stays open until its own input is complete. A memoized fingerprint is reused wherever its node is met again; for a node inside a cycle that depends on where the traversal first met it, so it is not that node's identity as a root, and only root fingerprints are compared.

11.5.1 Vectors

type graph, root first	fingerprint
`[0] Struct "Point" { x: [1], y: [1] }, [1] i32`	`0d91791f5827efadfa8150af9812fad6`
`[0] Enum "List", width 4, { 0 "Nil": [2], 1 "Cons": [3] }, [1] i32, [2] Struct "Unit" {}, [3] Struct "Cons" { "0": [1], "1": [0] }`	`abdbb8e6f2b104819c9b608dd77aaedf`
`[0] Opaque "lib" "T", version 1.2.3, fixed size 8, no parameters`	`81a2a3d8b939f4f216a344cf10758953`
`[0] Map { key: [1], value: [2] }, [1] Text, [2] u32`	`6a33dc4a1e560e465f7a50543f62c492`

Hex, byte 0 first. The Point's hash input, for instance, is `80 02 01 78 00 <16 bytes of i32's fingerprint> 01 79 00 <the same 16 bytes>`, and `i32`'s is the single byte `06`.

11.6 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.

11.7 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 types encode exactly as the writer's did, and decodes the sequence natively 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 *older* 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 for values the older reader cannot decode into its own types.

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 decodes a fresh in-memory value from the sequence when it opens a file, 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 "decode it from the writer's field *X*" or "use a default", and for each of the writer's fields the reader lacks, "skip it". Encodings are variable-size, so the plan is a parse of the writer's encoding rather than a list of offsets; it runs during the decode the open does anyway. Every value then follows the cached plan, with no string comparison and no graph walk.

A reader must be able to decode by the writer's descriptors. Every encoding is decodable from the descriptors alone, so a reader can skip a value it does not know, and read one whose type changed, by the writer's graph. 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.

The whole file decodes by one schema at every commit. The header names one schema chain and one root fingerprint, so a session that opens a file with a different schema either **upgrades** everything whose encoding changed before its first commit, or **retains** the writer's schema, writing in it, all session.

Upgrading writes the bytes of what changed, and nothing else. Say a file was written while `Shape` had two variants, `Circle` and `Path`, and the application opening it declares a third, `Rect`, and has given `Path` a new field. At open, the reader decodes every shape by the plan into its own `Shape`, and compares each shape's bytes in the file with that shape's encoding by its own schema. A circle's are equal, since adding a variant changes none of the old variants' bytes, so it keeps them; a path's differ, since its encoding now holds the new field, so the upgrade splices the new encoding in over the old one. Values whose type is the same in both schemas, such as a title, are not encoded at all; nor are containers whose types differ only through `Shape`, so the comparison passes through them to their shapes. Each splice states a constant number of spans besides its new bytes, and nothing behind a changed value is restated, although all of it moves, since no position is stored on file. The upgrade is a flush at open that writes these splices along with the new schema chain, before the application gets the file, and needs no journal: a crash before it commits leaves the old file, and the next open upgrades again.

Retaining interprets the writer's descriptors at every write. To write in the writer's schema, a writer needs the writer's encoding of each value it writes, and the positions of fields after a variable-size one, which come only from interpreting the writer's descriptors on the in-memory value at each write; only inside `Fixed` values do the descriptors give offsets directly. Upgrading is therefore the natural default, and retaining a mode for files that an older application must still read.

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

12.3 What is TBD

Upgrade or retain: the default, and how it is chosen. Is it a per-type attribute, a per-open flag, or decided by inspecting the diff? Upgrading is simple but breaks readback by the older application; retaining preserves that but complicates the write path. One should be the default and the other should be available.

Parts of a file in different schema versions: capsules. Keeping parts of a file in different versions of their schema, and upgrading each part on its own, is what *capsules* were sketched for, and one sequence makes them cheap. Three ideas made them up:

- a `Capsule<T>` is a schema boundary: a subtree that records which version of `T`'s schema it is written in, as an index into a table of the versions the file holds, each stored once and resolved once per session, however many capsules use it;
- fingerprints computed outside a capsule erase its inner type, so the root fingerprint does not change when `T` does, and a stale capsule is found by comparing the table's fingerprints with the reader's, once per version;
- each capsule upgrades on its own, at first touch, or keeps its version for as long as nothing writes it.

In one sequence a value's size concerns nothing but positions in memory, so a capsule is inline: its version's index as a varint, then `T`'s encoding in that version. Upgrading one at first touch is a splice of its region, with the new index, in the same transaction as the write that touched it; the capsule's handle, or whatever a binding has in its place, is where that happens. The schema chain would hold the table of versions in use: a session whose types are missing from it adds them in the flush at open, before any

capsule names them, and a count of capsules per version, built at open, says when a flush may drop a version. Nothing in the span format or the journal would change: capsules are a matter of the schema and of one wrapper type, and can come later. They would let a large file whose element type changed upgrade a part at a time rather than every element at the first open.

How much semantic-migration machinery to expose. Structural resolution handles add, remove, reorder, 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 vtables, 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.

Kladde's encoding is close to Avro's binary encoding too — no tags, no padding, zigzag varints, a union's branch as a varint, counts before sequences — and differs where mutation asks it to: Avro writes data once and reads it in sequence, while kladde edits values in place, so it keeps a sequence's count in one place rather than in blocks, ends text in a terminator so that an edit of text is one splice, and offers `Fixed` for values that change often.

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. kladde-rs's `kladde-inspect` reports on any kladde file at all three levels below, without consolidating or exporting it; kladde-cpp's `kladde-store-dump` prints a file's sequence, fingerprint and schema.

13.1 The premise

A kladde file is self-describing in two independent ways. It describes its **storage**: the span table is reachable from the header and describes every span of every chain, 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 schema chain says how every value in the sequence is encoded.

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, span-table and sequence layers. Requires **no** schema and no application knowledge.

A tool at this level can:

- list every chain, its length, and its spans with the pages they show bytes of;
- compute the live bytes, the description's share, the file's length, and therefore **garbage**;
- report the per-page live fraction, and how much of the file is superseded pages awaiting reuse;
- report the journal's length, the payload pages it names, 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 the content of no chain, 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*.

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 whether it is fixed-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 decoding the sequence by the descriptors. Requires no application knowledge.

A tool at this level can render the whole content as text by decoding the sequence from the root descriptor, front to back: primitives decode by their code, structs by their fields, enums by their discriminant, sequences and maps by their count, text up to its terminator. Every container is described structurally, there is no pointer graph to follow, and nothing can be unreachable, so the export is complete.

It stops only at **Opaque** types, whose content is by definition not decomposable: a tool can report their identity and version, and dump their bytes, which the encoding delimits, but not interpret them.

13.5 A canonical textual representation

The specification should eventually define a canonical textual representation of kladde files, and canonical conversions to and from common human-readable formats such as JSON. These representations would cover the schema and the sequence: not the private chain, which any writer may keep, replace or clear without changing what the file holds, and not how either is cut into spans. Two files with the same schema and sequence would then have the same text, whatever implementation wrote them and however often they were consolidated.

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.
- **Content.** Keep the sequence the canonical encoding of one root value at every transaction boundary, and refuse a file whose sequence is not.
- **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.** Group records so that every transaction takes a sequence that decodes to one that decodes (ordering).
- **Cuts.** Paste only cuts of the same journal, each at most once, and refuse a journal that does otherwise (cuts).

- **Bounds.** Support the stated bounds, and fail cleanly rather than wrap when an application exceeds them.
- **Entries.** Keep the rules for writers — one entry per span id and epoch, and the reach rule — and refuse a file that resolution finds invalid: a chain that names an id no entry describes, reaches a span twice, or shares a span with another chain, or a reached `Ref` outside a data page's content.
- **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, how content is cut across pages, and how it is cut into spans.
- 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 resulting sequence is the one an in-order replay produces.
- Assign, reuse, and recycle span ids however it likes, within the reach rule.
- Expose any API shape at all. Nothing above the storage layer is constrained.
- Keep the sequence in any in-memory form, or decode none of it: a tool that only compacts 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 private chain, 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 the content of any chain 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`.
- 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 private chain affect the sequence or the schema, or rely on a state found there without first checking that it is its own and still current (the private chain).

14.5 The suite

The conformance suite should have four 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 that its sequence decodes to the state after some prefix of the transactions. This is the mechanical check on the ordering rule: that every container groups the records of one edit into one transaction.
- *Payload pages.* For each of a set of mutation sequences with external bytes, drop every subset of the payload page writes since the last `fsync`, and assert that the file opens on the state after some prefix of the transactions, losing none that every write of its payload pages reached.
- *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.

Differential tests — an implementation checks itself against a model.

- *The store against a byte vector.* Any series of `Write`, `Splice`, `Move` and `Paste`, flushes, reopens and simulated power cuts must leave the sequence holding the vector's bytes.
- *The values against their encoding.* After every operation, the sequence must equal the canonical encoding of the in-memory root value, byte for byte.

No leak detector is needed: nothing in a file can be unreachable from the header.

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, an unassigned kind tag, or an undefined escape in the journal 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

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. The proposal of the file as one sequence settled that way: its content is now the specification's sequence and span table and the pages around them. So did the proposal of payload pages, which write a large transaction's bytes once: it is now the journal's external bytes, with their place in durability, the flush and the store's policy.

Most drafts below were written against the design it replaced, of allocations and an address table, and describe that design where they say "today"; their ideas about consolidation's policies carry over to pages of spans, and those about allocations, statements and places do not.

15.1 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

draft	status
Sparse leaves, and consolidation from the statements up	a finding, resolved: leaves stayed far emptier than data pages because consolidation interleaves choosing victims, deriving statements, and laying them out; the newest size wins ended the pile-up of nearly empty leaves, and dropping the fill floor on retiring leaves ended their staying half empty; and a redesign from the address table's logical level that keeps the three apart
Splices without restating the tail	a proposal for the address table: seven ways to state a splice without restating what lies behind it, measured on kladde-svg and on typing; recommends linked spans over backings, which the adopted design took further
A C++ implementation	a proposal for the high-level design of kladde-cpp: a describing macro now and reflection later in place of `#[derive(Durable)]`, enums from `enum class`, `std::variant` and class hierarchies, handles that outlive an edit, and four changes to the specification that a second language suggests

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 16

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.

16.1 Contradictions

Ordered most to least severe, where severity means *how likely this is to make someone write the wrong code*.

16.1.1 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 keeps the journal within a budget of journal pages: a transaction that would not fit is preceded by a flush, and only a transaction too large for the budget on its own exceeds it, taking as many pages as it needs, with a flush right after it.

16.1.2 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.

16.1.3 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.

16.1.4 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 predates the chaining, so it describes recovery as truncating a torn record rather than as keeping the longest valid prefix.

16.1.5 5. `mentions` is claimed to have one reader and has at least two

Liveness states flatly: "This is the only reader of `mentions`", meaning the release of a tombstone's pin at `mentions == 1`. In-memory state repeats it as "one thing only".

But 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 a second reader the field is harder to remove than the text suggests, and anyone auditing whether it can be dropped will reach the wrong conclusion.

16.1.6 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.

16.1.7 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.

16.1.8 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.

16.1.9 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.

16.1.10 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.

16.2 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.

16.2.1 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.

16.2.2 2. ~~The byte encoding of journal records~~ — resolved

Was: the record set had semantics but no bytes.

Resolved by Records and Framing.

16.2.3 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.

16.2.4 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.

16.2.5 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.

16.2.6 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.

16.2.7 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 size restatement obligation on page rewrites, where a bug silently resurrects truncated data. Both deserve assertions from the first commit rather than tests added later.

16.2.8 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.

16.2.9 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.

16.2.10 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.

16.2.11 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's page budget** — see flush triggers;
- **consolidation constants** — see below.

16.2.12 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.

16.3 Ideas for later

Ideas from kladde-rs's working notes that still apply to the present design and have no page of their own yet.

- **Records that call back into a type.** Some edits are far smaller to describe than their bytes — "draw a black line from X to Y" on an embedded image, or "recompress this image" — and a record naming the type and the edit, replayed by the type's own code, would journal them cheaply. It gives up what the journal's type-agnosticism buys: replay without application code and without a type registry, so a reader that lacks the type could not recover the file. At most it could be an optional optimization whose effect is also recorded as bytes at the next flush.
- **A lint for unusual combinations of wrappers**, once segments and capsules exist beside `DurableBlob`: the derive could suggest `Capsule<Segment<T>>` for a field declared `Segment<Capsule<T>>`, say, and say how to silence the lint, `#[kladde(allow_atypical_...)]`.

Chapter 17

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.

17.1 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.

17.2 It keeps the byte-referenced-once invariant

Which is the point. That invariant forbids two *live* `Ref`s 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.

17.3 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 `Size`.

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.

17.4 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 `Size(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.

17.5 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 18

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.

18.1 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.

18.2 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.

18.3 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 19

Cleaning by ripeness

Status: a proposal, not adopted. It would replace the budget's choice of victims by score and the churn floor in Consolidation, and leave free filling as it is. Cleaning by ripeness evaluates an earlier version, which estimated one rate per page, and Ripeness with main's free filling this one, on kladde-svg's drawings and text-bench's texts.

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 pages opened for cleaning that hold survivors of ripe pages only, so that they stay full.

19.1 The problem

19.1.1 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.

19.1.2 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.

19.1.3 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.

19.2 The proposal

19.2.1 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

$$V = \kappa\left(\frac{T}{u_0} - \frac{1-x^*}{r}\right) + x^*\left(\frac{1}{u_0} + V\right), \qquad \text{with} \quad T = \frac{\ln(1/x^*)}{r},$$
$$V = \frac{\kappa\,(\ln(1/x^*)/u_0 - (1-x^*))/r + x^*/u_0}{1-x^*},$$

and setting $\mathrm{d}V/\mathrm{d}x^* = 0$ 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

r/κ	x^*, cleaning at	the myopic rule $(1-x)/x \geq r/\kappa$ would clean at
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.

19.2.2 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: $r\,a$ 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, $\kappa\,(1-x^*) = r\,x^*\,(1+u_0 V)$, so $1 + u_0 V = (\kappa/r)\,g(x^*)$ with $g(x) = (1-x)/x$, and waiting saves $\kappa\,a\,g(x^*)$. The page is ripe once that no longer covers the cost:

$$1 - x \geq a\,g(x^*) \iff \frac{a}{a+1-x} \leq x^* \iff h\Big(\frac{a}{1-s}\Big) \geq \frac{r}{\kappa},$$

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 $1 - x \geq a\,g(x^*(r_1/\kappa)) + b\,g(x^*(r_2/\kappa))$, 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 $r_2 \to 0$: $x^*(0) = 1$ and $g(1) = 0$, so the second term vanishes, and with it both problems.

19.2.3 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 $\ell\, e^{rk}$, 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

$$\mathcal{L}(\ell, r) = S_0 \ln \ell + r\, S_1 - \ell\, E(r), \qquad \text{with} \quad S_0 = \sum_k e^{-\beta k} L_k, \quad S_1 = \sum_k e^{-\beta k}\, k\, L_k, \quad E(r) = \sum_k e^{(r-\beta)k},$$

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

$$\ell = \frac{S_0}{E(r)} \qquad \text{and} \qquad \frac{S_1}{S_0} = \frac{E'(r)}{E(r)}\,:$$

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 stays 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, `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, by the budget or by free filling, the page rewrite, and the rotating window 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
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

$$I = \min\Big(\frac{h(z)}{\hat{r}},\ \frac{h(x)}{R_{\min}}\Big),$$

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

$$r_0 = \frac{\sum_i r_i^2\, a_i}{\sum_i r_i\, a_i}, \qquad a_0 = \frac{\left(\sum_i r_i\, a_i\right)^2}{\sum_i r_i^2\, a_i},$$

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 $\hat{r} = \ln(\texttt{content_size}/\text{coverage})/\text{age}$. 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 `at` 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 description defragmentation and free filling move part of a page, free filling only once nothing whole fits, and then part of the sparsest victim, which leaves it sparser still and so first in line to be taken whole.

19.2.4 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.

19.2.5 Ranking pages by ripeness

Rank every page below u_0 by its ripeness index $I = \min(h(z)/\hat{r},\ h(x)/R_{\mathbf{min}})$, 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:

$$\ln I(\mathtt{now}) = K + \beta \cdot \mathtt{now}, \qquad \text{with} \quad K = \ln h(z) - \ln \mathtt{rate} - \beta \cdot \mathtt{at}$$

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^(β·now)
    settled:  BTreeSet<(Key, PageNumber)>,   // ln h(x) − ln R_MIN; the index is constant
}

impl Ripeness {
    /// The page with the highest index, if it is ripe at price `e^(−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(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 adds $O(\log P)$ per changed page and per budgeted victim to the bucket queues' $O(1)$, which free filling keeps. **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 $\beta \cdot \mathtt{now}$ 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, $x \to 1$: $h(x) \approx (1-x)^2/2$ 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, $x \to 0$: $h(x) \approx 1/x$ 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.

19.2.6 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 pages one flush writes for cleaning, and so does the check that an offer of data pages fills the page it opens; an offer of leaves opens none, since what it restates joins the leaves the cut lays out anyway, and costs the budget the pages its restatements take.

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.

Nor does the controller lower κ after a flush that cleaned nothing at the price: while the pages are fuller than τ, a lower price could change nothing there either. Without that hold, a file that starts full drives κ to its floor within its first flushes, and takes one to two live sizes of writes to climb back. Free filling's victims do not count as cleaned: it takes them whatever the price, in nearly every flush, and counting them would let κ fall in every flush while the pages are fuller than τ.

19.2.7 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 %.

19.2.7.1 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 the two kinds of page that take survivors take them by two rules: a page the budget loop opens holds survivors of ripe pages only, and the room of a page the flush writes anyway takes whatever costs nothing to move there. A page opened for cleaning can avoid mixing at no cost, so it does; the room of a flush's own page could avoid it only by giving up victims it frees with no page written for them, which, measured, costs more than the mixing:

source	where it goes	case
survivors of ripe pages	pages the budget loop opens, each filled with survivors of many ripe pages	cold with cold: no hot content to wait for
survivors of any victim that fits, the sparsest first, ripe or not	the room of a flush's own page	mixed, but the victim is freed with no page written for it, and ripeness cleans the host once waiting no longer pays

Neither needs a counterfactual about pairs: a ripe page is judged by its own rate against a threshold, and a victim of free filling by whether it fits the room.

19.2.7.2 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; free filling fills that room instead, so no ripe page's survivors are split between a hot page and a cold one.

19.2.7.3 The flush's own pages take what costs nothing

The room left in a page the flush writes anyway goes to what free filling takes in Consolidation, whatever the price: in compaction mode the tail first, then whole data victims that fit, the sparsest first, then description defragmentation's candidates beyond its share, then as many of one victim's statements as fit. A victim freed there costs no page, so its net gain is the whole page, whatever its fill or its ripeness; ripeness decides only what the budget pays to clean.

This mixes survivors, cold by selection, with fresh content, and on the workloads measured the free cleaning pays for it. The survivors in the room of a page are a share of at most that room, about half a page on a flush's last page; once the page's fresh content has died, it is a mixed page of that share, which ripeness cleans as it cleans any page, once waiting no longer pays. Keeping survivors out of that room would mix less: the room could go to ripe victims whose survivors take at most θ of a page, a share below their own threshold, and then to the survivors of the previous flush's least-filled page, a whole statement at a time, the content closest in age to the flush's own. On kladde-svg's drawings, that would clean 0.01 to 0.15 victims per flush free where free filling cleans 0.8 to 1.4, would open more pages for cleaning, and would write a median 3 % more at the same target for as many live pages; at equal writes, it would keep a median 1.1 to 1.2 % more live pages, and up to 6 % more at $\tau = 0.75$, and on text-bench's texts about as many (Ripeness with main's free filling).

Free filling's moves do not count as losses, neither in its victims' estimates nor in the host's, which starts as the byte-weighted mix of fresh and moved content, as a new page does.

19.3 What would change in Consolidation

- Scoring a data page would give way to the ripeness index for the budget's victims, and for table pages as well. The budget would take its victims from the two ordered sets, at $O(\log P)$ rather than $O(1)$ a victim; the bucket queues and their sampling would stay, for free filling's choice of the sparsest data victim.

- 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.
- Free filling would stay as it is, and its victims would not count as cleaning when the controller decides whether κ holds.
- 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, and $R_{\min}$.

19.4 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**, $x \to 0$: $h(x) \approx g(x) \approx 1/x$, 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 $8\,h(0.8)/r = 0.22/r$, 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.

19.5 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, with free filling's survivors, is a 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.
- **Where keeping survivors apart would pay.** On kladde-svg's drawings and text-bench's texts, the victims free filling frees outweigh what its survivors cost the pages they join; a workload whose survivors make a larger share of the room, or freeze their hosts above their thresholds more often, could tip the balance the other way, which only measurement can say.
- **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 20

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 $R_{\min}$.

20.1 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.

20.2 The decision under a posterior

20.2.1 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: $r\,a$ 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

20.2.2 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. Where survivors go stays as in the drafts too: free filling takes whatever fits the room of the flush's own pages, ripe or not, at no cost to the budget and without counting as cleaning for the controller, and the pages the budget opens hold survivors of ripe pages only.

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.

20.3 One rate per page

20.3.1 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

$$p(L \mid r) \propto r^{L/\sigma}\, e^{-rE/\sigma},$$

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.

20.3.2 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

$$r \mid \text{losses} \sim \text{Gamma}(A_0 + L/\sigma,\ B_0 + E/\sigma),$$

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.

20.3.3 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.

20.3.4 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, $A \approx r\,x/(\sigma\beta)$. 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.

20.3.5 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.

20.3.6 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

$$\mu = \frac{\sum_i k_i}{\sum_i E_i}, \qquad v = \frac{Q - (n-1)\,\mu}{\sum_i E_i - \sum_i E_i^2 / \sum_i E_i}, \qquad Q = \sum_i \frac{(k_i - \mu E_i)^2}{E_i},$$

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

$$\mu = \sum_i w_i\, m_i, \qquad v = \sum_i w_i\,(v_i + m_i^2) - \mu^2,$$

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 $A = \hat{r}\,B$.

20.3.7 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.

20.3.8 Deciding on one rate

With $r = G/B$ and $G \sim \mathrm{Gamma}(A, 1)$, 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 $\Phi_A(B\kappa) \le g(x)$, 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)** $P(\gamma \ge 0) = P(G \le B\kappa\,h(x))$, 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 $I_b = h(x)\,B/\,\mathrm{med}(A)$: the drafts' index at the posterior's median rate, with $\mathrm{med}(A) \approx A - 1/3$ for $A \ge 1$.

Both make an uncertain page riper than its posterior mean does, and the less the page has lost, the riper. φ is concave, so $\Phi_A(m) \le \varphi(A/m)$ and $I_a \ge I_0$; and the median of a Gamma distribution lies below its mean, so $I_b \ge I_0$. 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.

20.3.9 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:

$$J \sim \mathrm{NegBin}\Big(A,\ \frac{B}{B + aH/\sigma}\Big),$$

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.

20.4 A draining share over a static one

20.4.1 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.

20.4.2 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

$$p(r, \pi \mid \text{losses}) = \sum_{j=0}^{n} w_j\, \text{Gamma}(r;\, A,\, B_j)\, \text{Beta}(\pi;\, p_0{+}K{+}j,\, q_0{+}n{-}j), \qquad w_j \propto \binom{n}{j} \frac{\text{B}(p_0 + K + j,\; q_0 + n - j)}{B_j^A},$$

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.

20.4.3 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.

20.4.4 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.

20.4.5 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.

20.4.6 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.

20.4.7 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.

20.5 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 `Ref`s 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

chunks of	losses	the single-rate draft	posterior mean	(a)	(b)	(c)	(c′)
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 %
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.

20.5.1 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.

20.6 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 `rho`; 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

20.7 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 21

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.

21.1 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.

21.2 The proposal

21.2.1 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
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.

21.2.2 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.

21.2.3 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, which takes any victim that fits the room of the flush's own pages, as in the ripeness draft, so that the loop does not open pages for what that room 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 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 that room.

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.

21.2.4 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.

21.2.5 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.

21.2.6 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.

21.2.7 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.

21.3 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.

21.4 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.

21.5 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 22

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.

22.1 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 `id`s, followed by the attribute lists of elements and by path data:

drawing	allocations	attribute lists, median size	path data, median size	`id`s, 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.

22.2 The proposal

22.2.1 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, `bool`s, 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 `char`s 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.

22.2.2 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 `char`s, 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 `char`s, 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.

22.2.3 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 `char`s. 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.

22.2.4 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 `id`s are small strings, the attribute lists of the table above, is then a `SmallPersistableVec<Attr>`, and holds its first attributes' `id`s 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 `id`s 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.

22.2.5 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 varint, 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 varint 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.

22.2.6 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.

22.3 What a mutation costs

The cost depends on whether the mutation changes the size of the value it touches.

22.3.1 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.

22.3.2 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, `O(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.

22.3.3 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.

22.4 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.

22.5 What else it touches

22.5.1 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 `id`s as `SmallPersistableString`s inline in them; short path data could become `SmallPersistableVec<PathSegment>`, its segments packed.

22.5.2 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.

22.5.3 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.

22.5.4 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.

22.6 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.

22.7 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.

22.7.1 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 `f32`	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 `f16` where exact, else `f32`	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 `zigzag(digits) × 16 + (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.

22.7.2 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.

22.7.3 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.

22.8 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.

22.8.1 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.

22.8.2 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.

22.8.3 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.

22.8.4 `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.

22.9 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.

Chapter 23

Sparse leaves, and consolidation from the statements up

Status: a finding, resolved, and a direction for a redesign. The consolidation policy of Consolidation, as kladde-rs implemented it, kept address-table leaves far emptier than data pages, so that 4 to 33 % of a file's live pages were leaves it would not have needed had they been as full as its data pages. Three causes did it, and each is gone. The newest size rule let the statements that kept nearly empty leaves alive die, and removed the reason the cut refused them; that left leaves half empty, so that fuller ones would have saved 2 to 15 % of the live pages. A fill floor on retiring leaves kept them so, and the budget loop no longer holds leaves to it, since retiring a leaf opens no page; leaves as full as the data pages would then have saved 0.4 to 6 % of the live pages. Since kladde-rs also prices a leaf at its coverage, where it priced it at 1.25 times, the leaves come out about as full as the data pages under main's churn floor, and fuller ones would save at most 2.8 %. The three causes had one root, which is not a tuning constant but the way the policy interleaves three decisions, which page to retire, which statements to write, and where to put them, so that each constrains the others. The second half of this draft starts over from the logical level of the address table and keeps the three apart.

23.1 The finding

Leaves now come out nearly as full as data pages: on every run but the small tiger, 62 to 69 % full against 72 to 82 %, and at most 38 % of them under half full. On kladde-svg's drawings, with svg-bench's edits keeping to one part of the drawing at a time, a journal of 4 pages, and main's consolidation at its default churn floor, at kladde-rs `714c602` on its branch `defrag-shifts`:

drawing	layout	table pages	pages needed	fill	leaves under half full	under a tenth full	data pages under half full	share of live pages
tiger	base	10.8	7.1	66 %	23 %	0 %	4.7 %	29 %
tiger	packed	5.4	3.7	69 %	20 %	1 %	6.8 %	27 %
tiger	small	0.5	0.2	34 %	88 %	6 %	6.3 %	3 %
coat of arms	base	25.7	16.0	62 %	29 %	0 %	1.6 %	28 %
coat of arms	packed	14.7	9.3	63 %	28 %	0 %	3.0 %	29 %
coat of arms	small	2.4	1.6	67 %	38 %	1 %	2.4 %	5 %
world map	base	131.3	81.4	62 %	28 %	0 %	0.3 %	27 %

drawing	layout	table pages	pages needed	fill	leaves under half full	under a tenth full	data pages under half full	share of live pages
world map	packed	80.0	50.5	63 %	25 %	0 %	0.4 %	23 %
world map	small	13.3	8.8	66 %	29 %	0 %	0.3 %	4 %

"Table pages" is their mean count, "pages needed" what their live statements would fill, "fill" the ratio of the two, and the last column the table pages' share of the live pages. The means are over the rows svg-bench records every 1000 edits, and the shares over histograms of every live page's fill taken at the same flushes, by scratch instrumentation that did not change what was written. Were the table pages as full as each run's own data pages, the files would carry 0.4 to 6 % fewer live pages, the most on the world map in its base and packed layouts. The small tiger is the exception to the ranges above: its statements need a fifth of a page, which the header mostly holds, and it has one leaf at a time, or none.

What kept leaves a few points emptier than data pages was their price, not a rule that kept them out. A leaf's restatements were estimated at 1.25 times its coverage, since restated statements were thought to lose their neighbours' delta encoding, so at the default churn floor a leaf was a victim only up to 40 % full, where a data page is one up to 50 %; restated on their own, they take 0.94 to 1.05 times its coverage. Cleaning edited drawings and texts measures every setting of main's churn floor and of the three ripeness branches at the price of their coverage, at kladde-rs `7eb2ec4`, under which the churn floor's leaves come out about as full as its data pages.

23.1.1 Before the fill floor went

At kladde-rs `e88fd47`, the same branch before it stopped holding leaves to a fill floor, leaves stayed far emptier than data pages, and most of them sat between a fifth and a half full: the cut's fillers took the sparsest into the room left in its last page, and the budget loop, which held the rest to a fill floor, took few of them. The address-table pages averaged 44 to 60 % full, against 73 to 82 % for the data pages, and 40 to 67 % of them were less than half full at any time:

drawing	layout	table pages	pages needed	fill	leaves under half full	under a tenth full	data pages under half full	share of live pages
tiger	base	13.4	7.2	54 %	51 %	0 %	4.3 %	34 %
tiger	packed	6.3	3.8	60 %	40 %	1 %	7.1 %	31 %
tiger	small	0.8	0.2	21 %	88 %	44 %	5.0 %	5 %
coat of arms	base	30.5	15.9	52 %	53 %	0 %	1.9 %	31 %
coat of arms	packed	17.0	9.3	55 %	48 %	0 %	3.3 %	32 %
coat of arms	small	3.4	1.6	48 %	66 %	4 %	1.8 %	7 %
world map	base	183.8	81.4	44 %	67 %	0 %	0.3 %	34 %
world map	packed	103.0	50.6	49 %	58 %	0 %	0.4 %	28 %
world map	small	18.3	8.7	48 %	65 %	0 %	0.3 %	6 %

Were the table pages as full as each run's own data pages, the files would have carried 2 to 15 % fewer

live pages, and 8 to 15 % fewer on the coat of arms and the world map in their base and packed layouts.

Nearly empty leaves no longer stayed: on every run but the small tiger, at most 4 % of the leaves were under a tenth full, and in the base and packed layouts at most 3 % were under a fifth. What stayed was the band between a fifth and a half full, which held 38 to 65 % of the leaves in the base and packed layouts, and holds 18 to 29 % since the fill floor went. The world map in those layouts showed that part alone: its leaves were never nearly empty, and the new size rule raised their fill by 1 and 3 points only, where dropping the fill floor raised it by 18 and 14.

23.1.2 Before the newest size won

At kladde-rs `795b3b9`, the same branch before it moved to the newest size rule, nearly empty leaves piled up: on the tiger and the coat of arms, 5 to 92 % of the leaves were under a tenth full, and leaves as full as the data pages would have saved 4 to 33 % of the live pages. The same runs, then and at `e88fd47`:

drawing	layout	table pages, then	fill, then	under a tenth full, then	live pages by `e88fd47`
tiger	base	15.8	48 %	12 %	−6 %
tiger	packed	13.1	32 %	40 %	−25 %
tiger	small	8.8	3 %	92 %	−32 %
coat of arms	base	37.2	46 %	5 %	−6 %
coat of arms	packed	24.8	42 %	10 %	−12 %
coat of arms	small	9.1	19 %	62 %	−11 %
world map	base	203.6	43 %	0 %	−4 %
world map	packed	122.4	46 %	1 %	−5 %
world map	small	24.0	38 %	16 %	−2 %

The data pages were as full then as at `e88fd47`, and as since, so the live pages saved are the leaves'. The edited tiger that the variable-size evaluation, on the kladde-docs branch `defrag-shifts`, reports closing with 41 table pages for 2,265 statements likely showed the same effect.

On the small tiger, the leaves piled up for a dozen flushes and then vanished at once. Its table pages climbed to 11 to 19 at the peaks of the cycle, holding between 100 and 2,800 bytes of live statements in all; almost no table page was rewritten until compaction mode turned on, and that mode then rewrote 11 to 18 of them within one or two flushes, after which the count fell to one or two and started climbing again. At `e88fd47`, it held at most two leaves at any flush, and since the fill floor went, one.

23.2 Why leaves stayed empty

Three causes kept leaves nearly empty until the newest size won, and one of them, the fill floor, kept them half empty until it went too.

23.2.1 The budget loop held a table offer to the data pages' fill floor

The budget loop took a page rewrite only when the restatements of its victims would fill a whole leaf to `1 − θ`, 95 %, as it takes data victims only when their survivors fill a whole data page. kladde-rs' `implementation-notes.md` recorded it: "a budgeted page rewrite opens no page of its own … the budget counts it as a page all the same, and holds it to the fill floor like a data offer."

For a data page the floor has a reason: evacuating victims into a page the flush opens for them costs that page, and a page that closes half empty is a page wasted. For a table page it has none: a page rewrite opens no page, since what it restates joins the flush's dirty set and the cut lays it out among everything else the flush states. Its cost is the restatements' bytes, which the churn floor already weighs against the page it frees.

The floor turned the cheapest victims into the hardest to take: an offer was made of whole leaves at most half live, best score first, and needed 95 % of a leaf in restatements. A file whose leaves held less than that in all, such as the small tiger, could never pass it, however empty its leaves were. Where the leaves held more, an offer passed only where whole leaves happened to add up to 95 % of a leaf: a leaf between two-fifths and a half full restates into 50 to 62 % of one, by the estimate of 1.25 times its coverage, so two of them did not fit together, and one alone failed the floor.

What the floor refused waited for the cut's fillers, which take leaves only into the room left in the cut's last page, sparsest first. That room was 1.6 to 2.3 KB on average in these runs, enough for one leaf up to 31 to 45 % full, so the sparsest leaves went soon, and those fuller than the room allowed waited for an offer that passed, which is the band the finding showed at `e88fd47`. The fillers rewrote 52 to 96 % of the leaves rewritten in each run, and the budget loop met a table offer that passed the churn floor but not the fill floor 87 to 1,283 times per run.

Held to the churn floor alone, leaves come out nearly as full as data pages, and so the floor went. Since kladde-rs `714c602`, and in Consolidation, an offer of leaves passes on the churn floor alone, and costs the budget the pages its restatements take rather than a page; a leaf reclaims its page less the room its restatements take, so that at the default churn floor it is a victim up to 40 % full. The finding above shows the result: in the base and packed layouts, the leaves are 62 to 69 % full where they were 44 to 60 %, the files carry 4 to 10 % fewer live pages, and the flushes write 3 to 8 % more bytes per edit, restating leaves that the floor had kept waiting.

23.2.2 Leaves were kept alive for the size alone

Before the newest size won, much of what kept nearly empty leaves alive was statements that fixed sizes, and those could not die while anything older about their id was physically present. On the tiger and the coat of arms at `795b3b9`, 37 to 56 % of the live statements in leaves under a tenth full were `Shrink`s and `Tombstone`s, and none a `Grow`; on the small tiger it was 95 %, and 97 % of its nearly empty leaves, counted at every flush, held nothing else live. Under the old size rule, a `Shrink` stayed its id's anchor, pinned, until a newer `Shrink` or `Tombstone` replaced it, since without it the extents of older statements still in leaves would raise the size again; and a re-allocated id's tombstone stayed the anchor of its new incarnation.

Under the new rule, the newest statement that states the size wins, so an older one loses its pin as soon as a newer one states the size, and the cut makes every content statement that ends at its id's size sizing, so that happens whenever the cut states the id's tail. A re-allocated id's tombstone dies once the cut states the new incarnation's size. A leaf that held only such statements then falls to zero coverage, and the cut unlinks it without rewriting anything. That alone, with the fillers' check below still in place, cut the small tiger's table pages from 8.8 to 1.9 on average, and the packed tiger's from 13.1 to 8.5.

23.2.3 The cut's fillers refused every leaf that named an id the flush touched

The leaves that the budget loop does not take are meant to go as fillers: whole table victims that fit the room of the cut's last page, which costs nothing. That room was ample: in 8000 edits of the small tiger, logged at every flush, it was 800 to 3,500 bytes in nearly every one, and every nearly empty leaf would have fit. But kladde-rs took a filler only if the flush touched none of the ids the leaf names, and almost none qualified: on the small tiger, of up to 13 leaves under a quarter full, 0 were eligible in every flush, and on the coat of arms in its base layout, 0 to 2 of 2 to 14.

The check read a leaf's ids by decoding every statement still encoded in it, dead ones included. So a leaf with 4 to 25 bytes of live statements still named 74 to 251 ids, while each flush of the tiger touches 43 to 48 of its roughly 120 ids, and some named id was always among them.

Since kladde-rs `def217a`, the fillers take any leaf the flush has not rewritten already. Neither that nor the dying size statements suffices alone: with the new size rule but the check, 3 to 17 % of the leaves of the tiger and the coat of arms in their base and packed layouts were still under a tenth full, and 66 and 47 % in their small layouts.

The check guarded against stating an id's size twice in one epoch, which the new size rule lets the cut avoid without it; nor could the dead statements it read ever be part of a conflict. The next two sections show both.

23.2.4 What the check guarded against

A filler's victim can hold the size statement of an id the cut has already laid out statements for, and the cut must then state that id's size again, while one epoch holds at most one size statement per id. Content restatements cannot conflict, since a filler restates only fragments its victim still owns, which the flush has not taken. And the size statement need not: the cut decides which of its statements states each id's size only once the fillers are known, since the sizing bit changes no statement's encoded length, and adds a `Size` to the last page only for an id none of whose content statements ends at its size. So the cut can take any leaf as a filler, and kladde-rs dropped the check in `def217a`, once it had moved to these rules. Under the old size rule, the check had a reason: the statement an id needed depended on which pages survived, a `Grow` if its anchor stayed and a `Shrink` to replace the anchor if a filler retired it, and one epoch could not hold both, so a filler chosen after the layout could change a statement laid out already.

23.2.5 Dead statements cannot block a rewrite

Dropping a dead statement never changes what the file resolves to; it only releases pins that newer statements hold on its account. A statement is dead when no fragment resolves through it and it is not its id's size statement, `pins == 0`, and the droppability rule is exactly that dropping such a statement leaves the resolution unchanged. What a dead statement can still do is keep a newer statement alive, as an example shows:

- At epoch 4, allocation 8 holds 40 bytes, stated by `Ref*(8, 0, 40, c)` in leaf `L3`.
- At epoch 6, the application freed it: `Tombstone(8)`, in leaf `L4`.

`Ref*@4` is now dead, but physically present, and so it holds the tombstone: `mentions(8)` counts both statements, and a tombstone keeps its pin until `mentions` falls to 1, since without it `Ref*@4` would be 8's newest statement, and 8 would exist again with its old 40 bytes. Dropping `L3` drops `Ref*@4`, `mentions(8)` falls to 1, and the tombstone may retire: nothing resurfaces, which is the order the pins enforce, the denied statement first and its denier after.

Where the flush touched the id, a dead statement can at most make one of the flush's own statements unnecessary. If the flush frees an id and states its `Tombstone`, and a filler drops the last older statement naming the id, the new tombstone has nothing left to deny: a redundant statement of a few bytes, which the next rewrite of its page drops, never a wrong one. If the flush allocates 8 anew, the new incarnation states its size and every byte below it, and dropping `Ref*@4` changes nothing the cut has decided.

So the dead statements a leaf still encodes are no reason to keep it. The check read them because it learned a leaf's ids by decoding the page, which lists dead statements beside live ones, rather than by asking which of the leaf's statements are live.

23.2.6 Compaction mode collected them, in bursts

Compaction mode offers the highest live page ahead of every victim, whatever its fill, and an offer holding it passes both floors. Once free pages below the tail pass a quarter of the file, the mode works down the file's tail, page by page; the nearly empty leaves that the other paths left behind went with it, and once the mode switched off, they piled up again. That was the sawtooth on the small tiger, and why its file held 8.8 table pages on average for 0.2 pages' worth of statements; with no nearly empty leaves left for the mode to collect, it held 0.8 at `e88fd47`, and holds 0.5 since the fill floor went.

23.2.7 Data pages are not hit the same way

Data pages almost never fall below half full: 0.3 to 7.1 % of them, against 40 to 88 % of the leaves at `e88fd47`. They escape the fill floor: a data offer that whole victims leave short takes one more victim and cuts its survivors at the page boundary, carrying the rest into the next page, where a table offer held to the floor could take whole leaves only. And free filling takes any whole data victim that fits the room of any data page the flush writes, where the fillers have the room of one leaf. Their histogram's one feature, 14 to 25 % of them between 50 and 60 % full, is the default churn floor's own threshold, which takes victims at most half live.

But the same interleaving shapes them: a data victim is taken only if its survivors fit the room of a page being written, or fill a budgeted page to 95 % together with other victims, so which pages are cleaned

depends on how their survivors pack, not only on what cleaning them is worth. Nothing measured here shows that doing harm on data pages; the redesign below removes it on both sides alike.

23.3 The underlying error

The policy makes three decisions in one interleaved pass, which page to retire, which statements to write, and where to put them, and lets each constrain the others.

- Whether a leaf was retired depended on whether its restatements filled a page, which is a question about where they go. That was the fill floor, which no longer holds leaves.
- Which leaves the cut could retire depended on the statements it had already laid out. The fillers' check, which enforced it, is gone.
- The statements for an id were decided before the set of retired pages was final, so a later victim could contradict them. Since the newest size wins, the cut settles which of its statements states each id's size once the fillers are known, and content restatements never could conflict.

Each of these was locally sensible, and the later ones were added to keep the interleaving correct, but together they made the cheapest victims unreachable. The new size rule removed the two that guarded correctness, and the third, a matter of policy alone, went once it was clear that it protected no page. The same pattern makes the policy hard to reason about at all: the budget loop, free filling, the cut's fillers, the rotating window, and compaction mode each choose victims under different constraints, and each constraint comes from a decision made elsewhere.

23.4 Possible solutions, from the statements up

This section ignores how consolidation and the flush after its fold work today, and starts only from what statements mean.

23.4.1 What a writer must guarantee

A file's address table is the set of statements in its live table pages, each at its page's epoch, and their resolution defines the fragment map: each id's existence, its size, and for each byte, the statement that wins it and so its content. Call the statements `S`, and the fragment map they resolve to `R(S)`. Statements may contradict each other across epochs, since the newest wins at every probe; within one epoch, they may not.

A flush at epoch `e` starts from the file's statements `S` and from the map `M` that the application's writes since the last flush define: `R(S)` updated by the fold. It ends with a new set of live pages, whose statements `S'` must satisfy `R(S') = M`, and whose data pages hold every byte `M` resolves into the file.

Consolidation is the choice of which pages to give up, and what to write in their place, so that this still holds. That is three tasks, each of which needs only the results of the ones before it.

23.4.2 Task 1: choose the victims

Choose a set `V` of pages to retire, table and data pages alike, the current header always among them, by what each costs and what it gains, and nothing else. Retiring a data page costs the live bytes it holds, which must be written again elsewhere, and retiring a table page costs the statements that must be written because it is gone, which task 3 infers and which are usually its live statements' worth. Either gains a page.

Nothing about packing or layout enters the choice, since any `V` is feasible: task 3 can always state the resolved map again, as shown below. So a page whose live content is nearly nothing is the best victim there is, whichever kind it is, and is taken first; a page of dead statements costs nothing at all. How to rank the rest, by a churn floor, by ripeness, or by any other estimate of what waiting saves, is a separate question, and so is how many pages a flush may retire, which bounds its work. Returning space to the file system belongs here too: preferring the highest pages, as compaction mode does, is a ranking, not a mechanism of its own.

23.4.3 Task 2: combine the victims with the fold and the header

What survives is the statements in the pages not retired, `S_keep`, and the data bytes in the data pages not retired; everything else that `M` needs must be written.

- **Data**: every byte of `M` that resolves into a retired data page, or into the flush's own new content, must be written into a new data page.
- **Statements**: a set `N` of statements at epoch `e` must be found such that `R(S_keep ∪ N) = M`. The retired header's statements are among those that went, so whatever it stated that is still true must be inferred again like anything else.

23.4.4 Task 3: write the result

Writing out is two independent policies: one packs the data bytes into new data pages, and one infers `N` and lays it out into new table pages.

23.4.4.1 Data pages

The bytes to write are known once task 2 is done, so packing them is a bin-packing choice alone: in key order, as today, or sorted by temperature first, as the survivor-classes draft proposes, so that content that dies soon and content that waits do not share pages. Their addresses are what `N`'s `Ref` statements then point to.

23.4.4.2 Statements, by inference

`N` is what a reader needs beyond `S_keep` to resolve to `M`, and it can be inferred id by id, by comparing what `S_keep` alone resolves to with `M`. `N`'s epoch `e` is newer than everything in `S_keep`, so every statement in `N` wins over every surviving one where both match. For each id:

- **Existence.** If `M` says the id does not exist, `N` needs `Tombstone(id)` exactly when some statement in `S_keep` still names the id; otherwise nothing. If `M` says it exists, `N` needs at least one statement for it whenever `S_keep`'s newest statement naming it is a tombstone, or none names it; the size rule below provides one.
- **Content.** For every byte below the id's size in `M`, the winner among `S_keep`'s statements, the newest that matches, either already gives `M`'s content, or `N` must cover the byte with a `Ref`, `Inline`, or `Zero`. A byte whose winner sat in a retired page has no winner left among those that won it before, so it is covered again unless an older surviving statement happens to give the same content.
- **Size.** The size is what the id's newest size statement states, so `N` needs one exactly when `M`'s size differs from what `S_keep`'s newest size statement states — because the id was resized, or that statement went with a victim, or the id is new. It is the content statement in `N` that ends at the size, made sizing, or `Size(id, size)` if there is none.

Such an `N` always exists, and never breaks the rules of one epoch. The plain solution states, for every id whose resolution under `S_keep` differs from `M` at all, `Tombstone(id)` if it no longer exists, and otherwise content statements covering all of `[0, size)`, the last one sizing: it states the size, and the content statements win every probe below it. Those are one size statement per id and content statements on disjoint ranges below the size, which is all that one epoch's rules ask. The inference above only leaves out of that solution what `S_keep` already says, so it obeys the rules too, and no choice of victims can make the vocabulary fall short.

Inferring the smallest `N` needs nothing beyond the fragment map and each id's size statement. The content half is local: the fragment map already records which statement each byte resolves through, and so which page, and a byte needs a statement in `N` only when that page is retired or the byte changed. The size half is local too: each id has one size statement, and it needs stating again only when the size changed or its page is retired.

Laying `N` out is then a packing problem alone. `N` is final when layout starts, so the header can take the hottest statements, as it does now, and the leaves the rest, in key order or sorted by temperature, every page full but the last, with nothing to revisit. The room the last page has left is not filled by choosing more victims after the fact; if it should be used, task 1 can plan for it, since the restatements of a table victim, nearly empty or not, cost about what its live statements take.

23.4.5 A spec change that would make it simpler still: statements keep their epoch

Let each statement carry its own epoch, so that consolidation can move a statement to a new page without changing what it means. Today a statement's epoch is its page's, so a statement moved to a new page is re-stamped as newest: it now outranks everything it used to lie beneath, and the writer must work out what else to state so that the rest still resolves the same way. That re-stamping is why retiring a table page means deriving its restatements rather than copying them: a restated statement changes its rank, so the cut must state content as resolved truth, merged with whatever else the flush states, and state the size and the tombstone again where the victim held them.

With an epoch per statement, retiring a table page is copying its live statements, unchanged, into the flush's pages. They resolve exactly as before, since no two statements change their order, and the dead ones are left behind, which by the droppability rule changes nothing either. Evacuating a data page copies its referrers the same way, each with its new address and its old epoch, since the bytes it points to are the same. Only the flush's own changes are stated at its epoch, so task 3's inference is needed only for the ids the application touched, against a state the writer knows in full, and the size statements in the header stop moving every flush.

What it would cost and change:

- **Bytes.** A page would hold one or more runs, each of one epoch, given as a varint below the page's own, and each sorted by `(id, offset)`, as a page is now; a page the flush writes from its own changes alone is one run, as today, and a page of copied statements has a run per epoch it copies from.
- **The rule of one epoch**, no conflicts within it, would speak of statements of one epoch wherever they lie; a writer keeps it by retiring the page it copies from in the same commit, so that the copy and the original never both count.
- **Open's merge** would merge runs instead of pages; it already relies on each run having one epoch and being sorted by key, and only the number of runs grows.
- **Validation** would add that no statement's epoch exceeds its page's.
- **What estimates read from a page's epoch**, the age of its content, would come from its statements instead, which is finer than today.

No rule of one epoch needs relaxing for consolidation's sake, with or without this change.

23.4.6 What this would remove

- **The cut's fillers**, since the statements are inferred once the victims are known, and nothing laid out is ever revisited.
- **The distinction between free filling, budgeted pages, and compaction mode as ways to choose victims**: one ranking chooses every victim, and the room left in pages is a packing matter.

23.4.7 Open questions

- **Whether an epoch per run pays for itself**: its bytes against the restatements and re-stamped header statements it saves, which svg-bench's drawings could measure.
- **How victims are ranked once packing no longer constrains them**, and how much a flush may retire, which bounds both its work and how far the file can fall behind; this is where ripeness, survivor classes, and the churn floor's price of space come back in. Without the fill floor on table offers, main's churn floor fills the drawings' live pages to about nine tenths at $\lambda = 0.25$ (the evaluation).
- **Whether the rotating window survives**: it restates a key range for description density, which in this frame is a victim too, a range rather than a page, ranked by what its restatement saves.

Chapter 24

Splices without restating the tail

Status: a proposal for the address table, comparing seven designs; it recommends linked spans over backings, with their compaction policy as the open question. Today a splice restates every statement behind its point, because statements are keyed by their offset in the allocation, and a splice changes the offsets behind it. A design that states a splice in a constant number of statements must keep content statements at keys that no splice changes, and say elsewhere in which order their ranges follow each other.

Of the designs that do, the recommended one keeps an allocation's content in **backings**, offset spaces that no splice shifts, and orders it by **spans**, statements that show a range of a backing and name the span after it. A splice states two or three spans, wherever it lands, and moves no content statement. An allocation that was never spliced has one implicit span and is described exactly as today, at no cost.

Measured on kladde-svg's drawings and on typing into a long string, spans would state 1 to 6 bytes per edit, where today's flushes restate 0.4 to 24 bytes per edit that lay in leaves, and write 1 to 16 bytes per edit of data again. The gain is large where edits keep splicing long, fragmented allocations: 3 to 7 times on the small layout's two larger drawings and the slotted world map, and 7 to 16 times for typing. On kladde-svg's packed layout, where most splices land in a shape's own short lists, today's flush restates their tails in a statement or two, and spans gain little: nothing on the tiger and the coat of arms, and a third on the world map.

The price is that spans accumulate. Every splice leaves two more, so a list that is spliced again and again is described by ever more spans until something compacts them: without compaction, the drawings end with up to 9,600 live spans, more than twice the 4,302 statements the small world map has in all. Every design that states a splice in constant size has this price, in one form or another, since the order of the content has to be recorded somewhere; the designs differ in where, and in what compacts it.

24.1 What a splice costs today

A splice restates every fragment behind its point, since each fragment's key is its offset. kladde-rs, at `e88fd47` on its `defrag-shifts` branch, states a shifted piece of the fold as the fragments its bytes lie in, each at its new offset with its data where it was: a statement per fragment, and a data byte moved by none. Runs of fragments whose statements would cost more than their bytes, at this flush and the shifts its history predicts, it writes again as one chunk instead (the kladde-docs branch `defrag-shifts` documents both in `impl/flush.md`, under "Shifted bytes: restated, or merged").

What that costs the file depends on where the restated statements lay. A statement that lay in the header is restated by every flush anyway, so restating it at a new offset costs nothing more. A statement that lay in a leaf is restated into the header, as hot content, where it displaces other statements into new leaves; that, and the bytes written again, are what a splice adds.

On svg-bench's edits of kladde-svg's three large drawings, with its default focused edits, a journal of four pages and main's consolidation:

drawing	layout	table pages written	restated from leaves	restated from the header	written again, as data	all bytes written
tiger	slotted	19.0	1.3	3.3	1.5	128
tiger	packed	12.7	0.6	1.3	1.3	87
tiger	small	7.0	0.4	4.1	3.6	92
coat of arms	slotted	29.7	2.4	4.3	1.9	175
coat of arms	packed	16.0	1.2	1.7	1.8	109
coat of arms	small	15.4	6.0	5.1	5.2	130
world map	slotted	47.0	11.1	4.5	3.7	216
world map	packed	28.6	3.6	2.1	3.2	162
world map	small	39.8	24.3	6.2	15.5	214

All figures are bytes per edit; restated bytes are the statements' estimated encoded size, 9 bytes for a `Ref`, 4 for a `Zero`, and 4 plus the payload for an `Inline`. The layouts are those of the variable-size evaluation on the kladde-docs branch `defrag-shifts`: slotted lists, packed lists of enums, and small values for every list.

On the small layout's coat of arms and world map, the statements that splices take out of leaves amount to 39 and 61 % of the bytes of table pages written; elsewhere to 5 to 13 %, and 24 % on the slotted world map. Those are the runs in which ordinary edits splice long allocations. The small layout keeps a shape's whole content inline in its parent's child list, so a drag, a recolor or an inserted path node changes the shape's size and splices the parent's list, which on the world map holds up to 39 KB. In the packed and slotted layouts, those edits splice only the shape's own lists, which are short, and the long child lists are spliced by duplicates, deletes and restacks alone; the slotted world map's are the longest, at 38 bytes per child.

Typing into a long string costs as much as the journal does. A scratch benchmark typed 100,000 keystrokes into one string of 64 KB or 512 KB: 85 % insert a character at the cursor, 10 % delete the one before it, 4.5 % move the cursor, nine times in ten within a window of a tenth of the text, which jumps every 5000 keystrokes, and 0.5 % paste 200 bytes. Per keystroke, the flushes restated 6.6 and 18.4 bytes of statements from leaves, and wrote 7.5 and 17.9 bytes of table pages in all, where the journal took 13.3. It is the workload piece tables were invented for, and the one in which a splice's cost grows with the length of what it splices.

24.2 What every design has to answer

Two questions separate the designs: where the order of an allocation's content is recorded, and what compacts that record when it grows. A statement keyed by offset records content and order at once, which is why a splice must restate it. Every alternative splits the two, and each records the order in its own way:

design	order is recorded by	a splice states	what accumulates
A. offsets, restated or merged (today)	each statement's offset	every statement behind its point	fragments, which merging writes again as one
B. chunked containers	the container's chunk index	the statements behind its point within one chunk	chunks
C. spans placed by offset	each span's offset	every span behind its point	spans

design	order is recorded by	a splice states	what accumulates
D. **linked spans over backings**	each span's successor	two or three spans	spans
E. spans keyed by dense positions	each span's key	one or two spans, and a tombstone per span removed	spans, tombstones, and key length
F. a tree of spans	each node's list of children	the path from a span to the root	spans
G. shift statements	the epochs of the statements and the shifts between them	one shift	shifts

What accumulates must be compacted, and compaction costs about what eager restating saved, unless it runs rarely or cheaply. That is the trade-off between write amplification and space amplification that log-structured storage makes between updating in place and compacting later: today's design restates eagerly and keeps the description compact, and every other design defers the work to compaction. The designs that defer win where an allocation is spliced many times between compactions, and a policy that compacts too eagerly gives the gain back.

24.3 The designs

24.3.1 A. Offsets, restated or merged: today

Every statement keeps the allocation's offset as its key, a splice restates what lies behind its point, and runs that cost more to describe than to write are written again. It needs no change to the format, and the merge rule already keeps short allocations cheap: on kladde-svg's packed layout, splices cost 1.9 to 6.8 bytes per edit, about what the recommended design D states there. It also keeps the description compact: a merged list is one `Ref` per page, however often it was spliced.

Its cost grows with what lies behind the splice point, in statements or in bytes, so it is worst for long allocations spliced in many places, which are exactly what packed and variable-size values produce as documents grow: 14.8 bytes per edit on the slotted world map, 39.8 on the small one, and 19.4 per keystroke typing into 512 KB. In memory, every splice re-keys the fragment map's entries behind its point, `O(fragments behind)` tree operations per flush.

24.3.2 B. Chunked containers: keep allocations short

A container that may grow long stores its elements in chunks of bounded size, each an allocation of its own, under an index of the chunks; a splice shifts only the tail of one chunk. It needs no change to the format, and it bounds the cost of a splice by the chunk's size rather than the container's, in the table and in memory alike. It is a rope at the level of the data type, which the specification's design directions already consider for clones.

It bounds the cost rather than removing it, and it puts the work into every container instead of into the store once. Packed vectors, small vectors, strings and maps would each need chunking, the derive macro would need to know about it, and the type descriptors would have to describe chunked layouts for readers in other languages. Every chunk is an allocation, with its id, its size statement and its pointer in the index, which is what the small layout set out to remove. A move of an element between chunks writes its bytes again, and values kept inline inside an element still shift everything behind them within its chunk.

A container could instead leave slack between its elements, so that a splice shifts only to the next gap, as a packed memory array does; that brings back the padding that packing removed.

24.3.3 C. Spans placed by offset

Content statements move to backings, offset spaces that no splice shifts, and the allocation is described by spans, each stating at which offset of the allocation a range of a backing begins. Spans resolve by offset as content statements do today, newest wins, so the format gains one level and no new kind of rule, and an overwrite never touches a span.

But a splice restates every span behind its point, and spans accumulate with every splice, so the replay states more than today on every run measured: 5 to 71 bytes per edit on the drawings, and 27 to 31 per keystroke. It wins only where an allocation has far more fragments than spans, and only with compaction eager enough to give the gain back. It is rejected.

24.3.4 D. Linked spans over backings: the recommendation

Content statements move to backings as in C, and each span names the span after it instead of stating its offset; a splice restates the span it lands in and states the new ones, two or three statements wherever it lands. That is a piece table, kept in the file: the spans are its pieces, and the backings its buffers. It is the per-allocation indirection that the specification's design directions name as the open path for splice-heavy containers, with the order kept as links where they sketch a rope. An overwrite is one content statement in the backing that holds the bytes, shadowing as today, so the two most frequent edits of a drawing editor, overwrites and splices, both cost a constant number of statements.

Identities are stable because they are not positions. A span is known by where its bytes begin, its backing and its offset there, which no splice anywhere in the allocation changes; the order lives in the successor links alone. A design that numbered pieces in order would have to renumber them, or draw numbers from a dense order, which is design E. And a piece that owned its offset space, with no start of its own, could only be split by moving the tail of its content into a new piece, restating it; so a span shows a range of a backing, and splitting it at any point states two spans and moves nothing.

Existence follows from reachability, so a span that leaves needs no tombstone. A span is part of the allocation while the chain from the head reaches it; a span no longer reached is dead with its statements, and nothing can make it reached again except a statement of the epoch that does, by the rule below. Dropping a dead span's statement, or a dead content statement in a backing no span shows, can therefore never change what the file resolves to.

A page rewrite restates what is live in the page, each statement at its own key, and nothing else. No key changes when the page goes, so no statement elsewhere changes its meaning, and the rewrite has no obligation beyond the one it has today, stating the size again, which here is stating the head again.

What it costs: a second level of keys wherever a structure is keyed by offset; a walk of each spliced allocation's chain at open; a writer's choice per splice between restating and linking; and a compaction policy for the spans that accumulate. The design in detail spells them out.

24.3.5 E. Spans keyed by dense positions

Spans are keyed by positions drawn from a dense order, in which another position always lies between two, as fractional indexing, Logoot and LSEQ draw them; the order of the keys is the order of the content, so spans sort and delta-encode as statements do today, and no chain has to be walked. A splice states the new spans at keys between their neighbours'.

But nothing says that a span has left except a statement at its key, a tombstone that must outlive every older statement physically present at that key, or an older span would come back. That is the predicate over physically present statements that the newest size rule removed for sizes, now for every span a splice removes. And keys grow where insertions keep landing in one gap: with midpoints in base 128, the replay's keys averaged 1.3 to 6.3 bytes on the drawings, up to 28, and 10 to 14 bytes when typing, up to 26, where LSEQ's strategies for choosing a key near one end of the gap would shorten them. Labels of fixed width instead need relabeling, which order-maintenance algorithms bound to `O(log n)` labels per insertion amortized, restating spans again. On bytes written, it states about what D states, 1.6 to 5.0 bytes per edit, because it saves the successor links and pays for tombstones and keys.

24.3.6 F. A tree of spans

The spans hang in a tree whose interior nodes list their children and the lengths below each, as a rope or a counted B-tree does; a splice restates its span and every node on the path to the root, to correct the lengths. The order is explicit, no chain is walked, and any offset is found in `O(log n)` without reading the rest.

That random access is what a rope is for, and kladde does not need it in the file: a file is read whole at open, and every read after that is served from memory. What remains is a cost of `O(log n)` restated nodes per splice, each listing up to a node's fanout of children, where D restates `O(1)` spans; it is the path copying of a copy-on-write B-tree, whose cost statements exist to avoid. The tree belongs in memory, as the index over a spliced allocation's spans, where it answers which span holds a given offset; D keeps it there.

24.3.7 G. Shift statements

A statement `Shift(id, at, removed, inserted)` says that the content behind `at` moved; content statements keep the offsets of their own epoch, and a reader replays each id's epochs in order, shifting older content as it goes. It writes least of all designs: the replay states 0.8 to 2.5 bytes per edit for the shifts on the drawings, and 0.3 per keystroke. Shifts compact themselves through consolidation: a shift is needed exactly while some byte behind it, in today's offsets, is won by a statement older than it, since dropping it moves only such statements, and only to offsets at or behind its point; once consolidation has restated the content behind it, the shift is dead. That test is decidable from live state, with the minimum epoch of the fragments behind each point.

But the meaning of every statement depends on every newer shift of its id, which makes three things hard.

- **A shift can never be restated at a newer epoch**, since it would then shift what lies between its old epoch and its new one. The header, which restates its content every flush, can therefore not carry a shift, and no page holding a live shift can be rewritten, unless every statement keeps its own epoch, the change Sparse leaves considers for consolidation's sake. Without it, retiring a page that holds a live shift means restating every older fragment behind the shift: exactly today's cost.
- **Opening becomes a replay**: statements of different epochs use different offsets, so they cannot be merged by key in one sweep as today, and each spliced id is replayed epoch by epoch into a tree of fragments.
- **A shift cannot reorder**: within an epoch, an id's shifts form one monotone map of offsets, so moving content forward or back, as a restack does, writes the moved bytes again, 0.2 to 3.3 bytes per edit in the replay.

A design whose statements' meaning depends on statements elsewhere is what the newest size rule just removed for sizes; this one would bring it back for content. It is not recommended.

24.4 The recommended design in detail

24.4.1 Statements

Two kinds of statement are new; every statement of a never-spliced allocation stays as it is.

statement	meaning
`Backing(id, b)`	in this page, the content statements that follow, up to the next `Backing` or the next id, belong to backing `b` of `id`, `b >= 1`
`Span(id, b, start, len, next)`	allocation `id` shows `[start, start + len)` of its backing `b` here, followed by the span that begins at `next = (b′, start′)`, or by nothing

Backings are an allocation's offset spaces: backing 0 is the one every allocation has, and the only one of an allocation that was never spliced. A content statement belongs to backing 0 unless a `Backing` precedes

it, so statements of backing 0 are encoded as today, at no extra byte. A backing's number means nothing but identity; a writer numbers them per allocation and may reuse a number once no span shows it, by the same argument that lets it reuse an id.

A span is known by its key, `(id, b, start)`, and its newest statement is its value. The span keyed `(id, 0, 0)` is the allocation's **head**, and it alone may be empty, for an allocation whose first byte lies in another backing.

24.4.2 Resolution

- **Existence**: as today.
- **Form and size**: the newest statement of an id that either states a size or is its head decides. A size statement makes the allocation **contiguous**: its content is backing 0 from offset 0 to that size, as today. A head makes it **spliced**: its content is the concatenation of the spans reached from the head, following each span's `next`, and its size their total length.
- **Content** at an offset of a backing: the newest content statement of that backing that covers it, as today.
- **A spliced allocation is invalid** if its chain names a key with no span statement, visits a span twice, shows a byte of a backing twice, or shows a byte that no content statement covers.

Shown twice is forbidden so that no two places of an allocation alias one byte, which keeps the byte-referenced-once invariant meaningful without refcounts.

24.4.3 Rules for writers

- **No conflicts within an epoch**: at most one statement per id that states a size or is its head, at most one statement per span key, and content statements per backing as today.
- **What becomes part of an allocation in an epoch is stated in that epoch**, which generalizes the coverage rule: a byte of a backing becomes part of a spliced allocation when a span reached at the end of the epoch shows it and none reached at the end of the epoch before did; and a span becomes part of it when it becomes reached. Relinking a span that was reached before needs no statement of it; reaching one that was not, or showing a byte that was not shown, does.

Without the second half, a span that left the chain could come back with an old statement: a span keyed `(id, 2, 40)` is cut out at epoch 5 and its newest statement, now dead, is dropped by a page rewrite; an older statement at the same key, from epoch 3, is still physically present; a writer that linked `(id, 2, 40)` again at epoch 9 without stating it would show epoch 3's span.

24.4.4 Encoding

```
statement     := id_delta (tagged_ref | tagged_zero | tagged_tombstone | tagged_size
                         | tagged_span | tagged_backing | inline)
tagged_span   := 252:byte offset_delta:varint len:varint next_b:varint next_start:varint?
                 ; next_b = 0: the last span, without next_start;
                 ; else the next span begins at offset next_start of backing next_b - 1
tagged_backing := 253:byte b_delta:varint
inline      := tag:byte offset_delta:varint payload:byte{(tag - 4) >> 1}   ; 6 <= tag <= 251
```

Statements sort by `(id, backing, offset)`, a span at its `start`, before content statements at the same offset; a span moves the offset cursor by nothing, and a `Backing` resets it. The tags need room: `Inline` payloads would take at most 123 bytes instead of 125, which frees tags 252 to 255, two of them for these statements. A span takes 5 to 9 bytes, by the replay's estimate about 8.

24.4.5 Open and memory

Opening resolves each backing as today's sweep resolves an allocation, and walks each spliced allocation's chain once its statements are merged. The sweep already merges statements by key, and the key gains a backing; spans collect per id, and when an id's statements are done, its chain is walked from the head, each span's range taken from its backing's resolved runs, and everything the chain does not

show left unpinned. The checks that make a file invalid fall out of the walk: a missing key, a revisit, a byte shown twice, a shown byte that no run covers.

In memory, the fragment map is keyed by `(id, backing, offset)`, and a spliced allocation keeps an index of its spans by position. The index is the rope of design F: a counted tree, which finds the span holding an offset and takes a splice in `O(log spans)`; for a few hundred spans, a vector with running totals, updated in `O(spans)` per flush, does as well. No fragment is re-keyed by a splice, so a flush no longer does `O(fragments behind)` work in memory for each. A reached span holds a pin of a third kind beside `F` and `S`, taken when the chain comes to reach it and released when it stops; the head holds `S` besides, as the statement that decides the allocation's form.

24.4.6 Flushing

The fold is unchanged; the flush maps its pieces through the span index. Committed bytes taken from an old offset become the backing ranges that show them; new content goes into a backing of the flush's own, one per spliced allocation and flush, or onto the end of the backing whose end the allocation's tail shows; overwrites in place go to the backings that hold the bytes. The flush states the spans whose length or successor changed, and the new ones.

A writer may restate instead of linking, and should while the tail is short. An allocation stays contiguous, its splices restating its tail as today, until restating would cost more than the two or three spans a link costs; the measurements below show where that line runs: short packed lists stay contiguous, long lists and strings become spliced.

24.4.7 Compacting spans

A spliced allocation is compacted by writing runs of short spans again as one, and by flattening it: restating its content in backing 0, at the allocation's offsets, and stating its size, which makes it contiguous again. Writing short spans again costs their bytes and saves their statements, the trade today's merge rule already makes, but once instead of at every later shift. Flattening costs a statement per content fragment the allocation has, today's restatement paid once instead of at every splice.

A rule that flattens an allocation once its spans outnumber, by a margin, the content statements it had after its last flattening keeps its spans about as few as its content statements, and pays for each flattening with the splices before it. On the typing runs, it cut the live spans from about 8,000 to 1,786 and 1,460, for 0.8 and 1.1 more bytes per keystroke. On the drawings it barely fired, because the replay keeps every content fragment an overwrite made, without the description defragmentation that kladders runs; how it behaves there is for an implementation to measure.

24.4.8 What it changes elsewhere

- **The newest size rule** holds for contiguous allocations as it is; for a spliced one, the head takes the size statement's place, and a size statement newer than the head makes it contiguous again.
- **Stating the size again**, a page rewrite's one obligation, extends to the head.
- **Description defragmentation** works per backing, and gains flattening as a way to pay down a description.
- **The reverse index**, from a data page to the ranges that point into it, the dirty set, and the consolidator state key ranges by backing as well as by id.
- **A move within an allocation**, as a restack, is a relinking that writes no data byte; the `Move` operation between allocations is unaffected.

24.5 Measured

All designs but C state far less than today where long allocations are spliced, and about as much where they are short; D pays for its constant cost with spans that accumulate.

24.5.1 How

Scratch instrumentation of kladde-rs `e88fd47` recorded every flush's fold, the piece table of every allocation it touched, and what today's flush made of the shifted pieces, without changing what was written; a replay then derived from the same piece tables what each design would state. The runs are those of the table above, and the two typing runs. The replay keeps each allocation as spans over backings, maps each piece table through them as the flush above does, and counts the statements whose key or value changed, at their encoded size with varints. For today, the bytes are what the flush restated from leaves and what it wrote again, from the instrumentation.

The replay is a model: it does not run consolidation, so it counts what flushes state, not where statements end up; and it models content fragments only as boundaries of writes, without description defragmentation.

24.5.2 What the flushes state

Bytes per edit, or per keystroke for the typing runs; "+" separates statements from data written again:

drawing	layout	today, from leaves + written again	D: spans	D, rewriting short spans	E: dense keys	G: shifts + moves written again	C: spans by offset
tiger	slotted	1.3 + 1.5	2.1	2.1 + 0.0	1.6	0.8 + 1.0	5.1
tiger	packed	0.6 + 1.3	2.1	1.7 + 1.1	2.1	0.8 + 0.2	6.9
tiger	small	0.4 + 3.6	2.5	2.2 + 0.6	4.3	1.3 + 0.8	10.9
coat of arms	slotted	2.4 + 1.9	3.1	3.1 + 0.0	2.2	1.2 + 1.3	8.2
coat of arms	packed	1.2 + 1.8	2.9	2.5 + 1.5	2.5	1.1 + 0.3	8.9
coat of arms	small	6.0 + 5.2	3.5	3.4 + 0.4	4.3	1.9 + 2.1	18.5
world map	slotted	11.1 + 3.7	4.2	4.2 + 0.0	2.4	1.7 + 1.7	17.8
world map	packed	3.6 + 3.2	4.3	4.0 + 2.5	2.9	1.7 + 0.4	25.2
world map	small	24.3 + 15.5	5.8	5.7 + 0.7	5.0	2.5 + 3.3	71.2
typing, 64 KB		6.6 + 1.3	1.1	1.0 + 0.2	2.3	0.3 + 0.0	26.7
typing, 512 KB		18.4 + 1.0	1.2	1.2 + 0.1	1.9	0.3 + 0.0	30.6

"Rewriting short spans" writes a run of spans again as one where its spans and their content statements, counted at 14 bytes each, outweigh its bytes, by today's merge rule over spans.

- **D's cost does not grow with the allocation**: 1.1 to 5.8 bytes per edit on every run, where today's grows from 1.9 to 39.8 with the length and fragmentation of what is spliced.
- **On the packed layout, D gains little**: 2.1 and 2.9 bytes per edit on the tiger and the coat of arms, against today's 1.9 and 3.0, and 4.3 against 6.8 on the world map, since most splices there land in short lists, whose tails today's merge keeps at a statement or two. That is why the writer should keep short allocations contiguous.
- **G writes least**, and its moves written again cost as much again on the small layout; what it would cost in compaction the replay cannot say, since that depends on when consolidation restates the content behind each shift.
- **C is worse than today everywhere**, since spans accumulate and it restates all of them behind each splice.

24.5.3 What accumulates

Live spans at the end of each run, against the statements today's file holds in all:

drawing	layout	today's statements	D, live spans	in the largest allocation	rewriting short spans	and flattening
tiger	slotted	1,752	738	142	738	626
tiger	packed	1,585	818	224	512	459
tiger	small	459	841	270	656	569
coat of arms	slotted	3,857	1,855	263	1,855	1,752
coat of arms	packed	3,529	2,058	284	1,421	1,385
coat of arms	small	1,130	2,081	630	1,739	1,664
world map	slotted	17,966	8,953	305	8,953	8,413
world map	packed	15,944	9,381	426	6,719	6,373
world map	small	4,302	9,631	787	8,394	7,833
typing, 64 KB		2,501	8,030	8,030	5,785	1,786
typing, 512 KB		5,608	8,477	8,477	7,830	1,460

Kept, spans would add about half again to the slotted and packed files' statements, and nearly triple the small layout's. The allocations with the most spans show why: one of 23 KB on the small world map ends with 787 spans, and one of 1.9 KB on the packed world map with 426, a span for every 4.5 bytes. Rewriting short spans helps where elements are short, in the packed layout, and not where they are 38 bytes, in the slotted one. Flattening paid off on the typing runs, at 0.8 to 1.1 bytes per keystroke, and fired too rarely on the drawings to judge, for the reason above.

24.6 What the literature does

Text editors keep sequences as piece tables or ropes, in memory; file systems key extents by offset, and pay for inserting in the middle; collaborative editors give elements identities independent of position.

- **Piece tables** describe a text as a sequence of pieces, each a range of an unchanging original buffer or an append-only one, so an edit changes the sequence and never the buffers; Bravo and Microsoft Word used them, Charles Crowley's "Data Structures for Text Sequences" (1998) compares them with gap buffers and others, and Visual Studio Code's piece tree (2018) keeps one in a balanced tree. Design D is a piece table whose sequence lives in the file as a linked list, and whose index lives in memory.
- **Ropes**, Boehm, Atkinson and Plass's "Ropes: an Alternative to Strings" (1995), are balanced trees over string fragments for fast concatenation and indexing, an in-memory structure; counted B-trees and finger trees serve the same purpose.
- **Extent trees of file systems** key a file's extents by offset, as today's statements do: inserting a range, as Linux's `FALLOC_FL_INSERT_RANGE` on ext4 and XFS does, shifts every extent behind it, the cost kladde pays today. Copy-on-write B-trees, as Ohad Rodeh's "B-trees, Shadowing, and Clones" (2008) describes them, copy the path to the root on every change, which is design F's cost.
- **Sequence CRDTs** give every inserted element an identity independent of its position, and order elements either by links to a neighbour, as RGA and Yjs do, or by dense position identifiers, as Logoot and LSEQ do; they keep tombstones for removed elements until every replica has seen the removal. A single writer needs none of their machinery for merging concurrent edits, which leaves links, design D, and dense keys, design E.
- **Order maintenance**, from Dietz and Sleator (1987) to Bender et al.'s "Two Simplified Algorithms for Maintaining Order in a List" (2002), keeps fixed-width labels in order by relabeling `O(log n)` of them per insertion, amortized.

24.7 Open questions

- **The compaction policy**: when to rewrite short spans, when to flatten, and how much description a hot list may carry; it is the policy this design adds, and the one a test should probe for failure modes before anything else.
- **Where the line between restating and linking runs**, which the replay places between kladde-svg's packed lists and its long lists, but which a writer has to find per splice, from the tail it would restate.
- **Whether the head and the size statements should be one kind**: both decide an allocation's form and size, and the newest of them wins; a size statement could be read as a head over backing 0 that runs to the size, which would leave one rule where there are two.
- **What the in-memory index costs**, in memory and in time, against today's `O(fragments behind)` re-keying per splice, which this draft does not measure.
- **Whether shift statements' low write cost is worth a second look** once statements keep their epoch for some other reason; without that, they are not an option.

Chapter 25

A C++ implementation

Status: the design of kladde-cpp, now built. kladde-cpp records it as built, with the places where building it went another way, and Porting kladde to C++ and into Inkscape what building it found; the decisions below are settled, and kept for their reasoning. It answers two questions that were open from the start — what stands in for Rust's `#[derive(Durable)]`, and how C++ types map to the schema's enums — and raises a third that turned out to matter as much: how long a handle to a value may live.

The specification is not frozen, and finding out what a second language needs from it is half the point of a second implementation. So this draft does not take the specification as given: where a change to it would make kladde-cpp simpler or its API more idiomatic, Changes to the specification proposes it, with what it costs the format and kladde-rs. Four candidates came up, besides a few smaller points; one of them, on how an enum's variants hold their fields, changes how C++ maps enums enough that the sections on enums assume it, and say what they would do without it.

Like every draft, it mixes what belongs in kladde-rs's counterpart of a language section with what may turn out to be a proposal for Implementation or Specification; the open questions at the end say which parts might travel.

25.1 What drives the design

A library for many applications, with Inkscape as the first. The first demo is a fork of Inkscape that edits a kladde file with an SVG schema and has no save button: every dragged control point and every keystroke is durable when the call that made it returns. So the design has to suit an existing, large, object-oriented C++ code base that holds pointers to its document's nodes for as long as it likes, and not only programs written for kladde from scratch.

C++20 as the baseline, C++26 reflection as the destination. Inkscape builds with C++20, and the compilers of today's long-term-support distributions (GCC 13 on Ubuntu 24.04) know nothing of reflection. Static reflection is in C++26, and compiler support for it is arriving (GCC 16 has an implementation behind `-freflection` at the time of writing), so the design must work without it and get better with it, without a second API.

Idiomatic C++, not Rust transliterated. The overview asks for this, and the differences that matter are few but deep:

	Rust	C++
deriving code from a type	procedural macros see the type's tokens	the preprocessor sees tokens it cannot parse; C++26 reflection sees the type
exclusive access while mutating	enforced by the borrow checker	a convention, checkable at run time in debug builds
long-lived references into a structure	fight the borrow checker	ordinary, and expected by existing code

	Rust	C++
trait objects	need an enum and a `dispatch` attribute to be durable	class hierarchies with virtual functions are the native idiom
errors	`Result`	exceptions, which Durability already names for C++

Everything below the typed values is the reference algorithms, ported. The storage engine — pages, span table, rope, journal, flush, consolidation — is language-agnostic by construction, and Implementation describes it closely enough to port. Most of the effort of kladde-cpp is there, and C++ asks almost nothing of it that Rust does not; the exceptions are a flush that does not block the caller and values detached for longer than a transaction, both of which an interactive document editor needs more than C++ does. This draft is mostly about the part on top, where C++ and Rust part ways.

25.2 Three layers

layer	holds	form
the store	the file format, durability, journal, flush, consolidation; positions and bytes only	a compiled library with a small non-template interface
the schema	descriptors, their canonical encoding, fingerprints, and later resolution	compiled, plus `constexpr` helpers that build descriptors from types
typed values	which C++ types are durable, how they encode, and the handles that mutate them	header-only templates over the store's interface

The store knows nothing about types, exactly as in the reference algorithms: it takes `write`, `splice`, `move` and `paste` at positions in the sequence, opens and ends transactions and batches, and flushes. Keeping it free of templates keeps compile times and code size of applications down, keeps the hard part testable on its own against a plain byte vector (the differential oracle), and gives it a C ABI later almost for free, should another language want to bind it.

Why a native port rather than binding kladde-rs. Linking the Rust store through a C ABI would be quicker to a first demo, and is a fair fallback if the port lags. But a second, independent implementation is how the specification finds out where it is ambiguous, which is worth more at this stage than the code it saves; and a C++ library that needs a Rust toolchain to build is a hard sell to the applications it is for.

25.3 The shape of the API

```
#include <kladde/kladde.hpp>

struct Contact {
    std::string email;
    kladde::Vector<PhoneNumber> phones;
    bool starred = false;
};
KLADDE_STRUCT(Contact, email, phones, starred)

void example() {
    auto file = kladde::File<Contact>::create("contacts.kladde",
                                              Contact{.email = "ada@example.com"});

    Contact const& contact = file.get();   // reads: the plain value, no cost
    std::size_t n = contact.phones.size();
```

```
    auto c = file.edit();                  // kladde::Mut<Contact>
    c.starred().set(true);                 // a Write of one byte
    c.email().insert(0, "lady.");          // one Splice
    c.phones().push_back(Mobile{"555-0100"});  // one transaction
}
```

`kladde::File<T>` is the root, Rust's `Kladde<T>`: it owns the value and the store behind it, and is movable but not copyable. `get()` hands out `T const&`, so reading is plain C++: members, iterators, range-for, algorithms.

`kladde::Mut<T>` is the handle that mutates, the guard of kladde-rs under a name that says what it is for. It holds a pointer to the in-memory value and the value's region of the sequence, and every mutating method records first and changes memory second. It dereferences to `T const&`, so reading through a handle works as reading the value does. The only way to mutate a value inside a `File` is through a `Mut`, as in Rust; C++ cannot forbid `const_cast`, but it does not have to make it easy.

25.4 The analog of `#[derive(Durable)]`

What the derive gives a Rust type is five things: the encoding, the decoding, the descriptor, the fixed size, and the guard with an accessor per field. The first four follow mechanically from the list of fields — their names, their types, and how to reach them — and the fifth needs names that the program can call. So the question is how C++ gets at the list of fields.

25.4.1 The options

option	C++	how it looks	verdict
hand-written traits	any	specialize `kladde::Durable<T>` by hand	always available; the escape hatch, as in Rust
a describing macro	20	`KLADDE_STRUCT(Contact, email, phones, starred)`	**the baseline**
member enumeration without a macro	20	nothing, for aggregates, by the tricks of Boost.PFR	too fragile for something that decides a file format
code generation from a schema file	any	an IDL and a generator, as Protocol Buffers do	wrong default for a code-first library; right for a later tool
static reflection	26	`struct [ [=kladde::durable] ] Contact { ... };`	**the destination**

Recommended: one internal description of a type, `kladde::describe<T>`, fed by a macro today and by reflection later. Everything else in the library — encoding, decoding, descriptors, fixed sizes, handles — is written once against that description, so a type described by the macro and a type described by reflection behave identically and have the same fingerprint, and an application can switch per type when its compiler allows.

25.4.2 The macro

`KLADDE_STRUCT` follows the pattern that C++ libraries have converged on for exactly this job, Boost.Describe's `BOOST_DESCRIBE_STRUCT` and nlohmann/json's `NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE`: it sits next to the type, in the type's namespace, and lists the members to persist, in encoding order. It expands to two things, found through argument-dependent lookup so that the user never opens namespace `kladde`:

```
// Roughly what KLADDE_STRUCT(Contact, email, phones, starred) expands to.
constexpr auto kladde_describe(kladde::type_t<Contact>) {
    return kladde::struct_description("Contact",
        kladde::field<&Contact::email>("email"),
```

```
        kladde::field<&Contact::phones>("phones"),
        kladde::field<&Contact::starred>("starred"));
}

template <class Self>   // mixed into kladde::Mut<Contact>
struct kladde_accessors_Contact {
    auto email()   { return static_cast<Self&>(*this).template at<&Contact::email>(); }
    auto phones()  { return static_cast<Self&>(*this).template at<&Contact::phones>(); }
    auto starred() { return static_cast<Self&>(*this).template at<&Contact::starred>(); }
};
template <class Self>
kladde_accessors_Contact<Self> kladde_accessors(kladde::type_t<Contact>, Self*);
```

The description is a `constexpr` list of member pointers and names, from which templates compute the encoding, the decoding, the descriptor and the fixed size at compile time. **The accessors give `c.email()` its name**, which is what Rust's `email_mut()` is, with completion in any editor that expands macros. Where a field's name collides with a method of `Mut` itself, such as a field named `set`, the member pointer reaches it instead: `c.at<&Contact::set>()`.

`auto [email, phones, starred] = c.parts();` hands out the handles of all fields at once, as Rust's `parts()` does, through the tuple protocol that structured bindings use.

A second form goes inside the class, `KLADDE_MEMBERS(email, phones, starred)`, for types whose persisted members are private: it defines the same two things as a hidden friend and a nested template, which may name private members.

What the macro buys over Rust's derive, besides working at all:

- **Members it does not list are not persisted.** A cache, a handle, a pointer to a widget stay in the C++ type and out of the file, with no hand-written implementation; Rust's derive has no attribute for this yet.
- **Field names on file can differ from names in C++**, through a longer form, `KLADDE_FIELD(member, "name")`. Field names are fingerprinted and are what evolution reconciles by, so this matters for files that applications in two languages, with two naming conventions, share.
- **Options ride along as arguments**: a field held in its fixed form, a single-field struct marked transparent, a struct marked bounded, as `#[kladde(...)]` attributes do in Rust.

What it costs is that the member list repeats the declaration, and nothing checks that it lists every member that should be listed. That is the price C++20 asks of every library of this kind, and the reason to move on as soon as reflection allows.

25.4.3 Reflection

With C++26, the description comes from the type itself:

```
struct [ [=kladde::durable] ] Contact {
    std::string email;
    kladde::Vector<PhoneNumber> phones;
    [ [=kladde::fixed] ] std::uint64_t revision = 0;
    [ [=kladde::skip] ] Cache cache;   // not persisted
};
```

Annotations are written `[ [=…] ]` on these pages, which C++ accepts as readily as the usual form without the spaces, because the site would take the usual form for a link. The annotation is the opt-in, as `#[derive(Durable)]` is: a file format should not change because someone added a member to an unrelated struct. `kladde::describe<T>` walks the non-static data members with `std::meta::nonstatic_data_members_of`, takes their names from `identifier_of`, and their options from annotations.

Reflection cannot generate member functions, so `c.email()` stays the macro's; but `std::meta::define_aggregate` can generate a struct with a data member per field, which gives `c.parts().email` with its name, and that is the form to recommend for reflected types.

25.4.4 Generics, and what is durable

A class template is described once, with a variant of the macro that takes its template head, and is durable wherever its parameters are. Lifetimes have no counterpart in C++, and non-type parameters can simply be allowed, since the description is computed per instantiation.

25.5 Which C++ types are durable

Standard types are durable wherever the in-memory value needs nothing that a standard type cannot hold, and kladde's own containers take over where it does. Rust refuses `String` and `Vec` to make durability visible in the type; in C++ the handle, `Mut<std::string>`, is a type of its own whatever the value is, so the plain type costs nothing in clarity and lets existing types persist unchanged.

@Claude: the rust API has grown over many iterations with vastly different designs of kladde. An earlier design of kladde made it impossible to use `String`, `Vec`, or `HashMap` directly in a durable struct, which motivated the dedicated types `DurableString`, `DurableVec`, `DurableHashMap`, and possibly others back then. But I'm not sure if these dedicated types are still necessary in today's design of kladde. Which of the `Durable` types that kladde-rs defines are still necessary and which could be replaced with their standard rust types without cost?

Claude: `DurableString` could be replaced by `String` at no cost, `DurableVec<T>` by `Vec<T>` for fixed-size `T` only, and `DurableMap` and `DurableBlob` are still necessary.

- **`DurableString` is a `String` and nothing else** (`struct DurableString(String)`), and its guard is a type of its own anyway, so `impl Durable for String` in kladde-persist, where the trait lives, would give the same encoding, the same guard and the same cost. The only argument left for it is the one the tutorial gives, that a field's type should say it is durable, which a guard of its own already says.
- **`DurableVec<T>` is a `Vec<T>` plus a Fenwick tree of the elements' sizes**, and the tree is empty for a fixed-size `T`. A `Vec<T>` has nowhere to keep the tree, so it can stand in only where no tree is needed; Rust has no specialization, so that would take a marker trait implemented by every fixed-size type (the derive can implement it where all fields are fixed-size, as it implements `Bounded`), and `impl<T: FixedSize> Durable for Vec<T>`. For variable-size elements a `Vec<T>` would have to find element `i` by summing the sizes before it, `O(m)` per guard, which is what the tree exists to avoid.
- **`DurableMap` is necessary**: a `HashMap` iterates in an order that changes when it rehashes, while the file keeps entries in one order that each entry's position depends on, and neither `HashMap` nor `BTreeMap` can keep a Fenwick tree of entry sizes by rank.
- **`DurableBlob` is necessary as an opt-in**: `impl<T: Serialize> Durable for T` would conflict with every other implementation, so a serde type needs a wrapper to say "persist me through serde"; it is also where the serialized bytes are cached.
- `Option`, `Box`, tuples and the primitives are already the standard types, and `Fixed<T>` is a choice of encoding that only a type can carry.

Decision: remove `DurableString`; implement `Durable` for `Vec<T>` where `T: Durable` with a const assertion that `T::FIXED_SIZE.is_some()`. Update the documentation and any examples in the tutorial, if applicable. `DurableVec<T>` stays and should work on all `T: Durable`, even fixed-sized ones. For fixed-size `T` it should not allocate a Fenwick tree (but it still has the field for the empty Fenwick tree).

So kladde-rs could follow the C++ table: `String` for text, `Vec<T>` for fixed-size elements, and a dedicated vector for variable-size ones.

C++	kind	note
`std::uint8_t` ... `std::uint64_t`, `std::int8_t` ... `std::int64_t`	Primitive	matched by width and signedness, so `long` maps to whichever it is
`float`, `double`, `bool`, `char32_t`	Primitive	`char32_t` is the schema's `char`; plain `char` is refused, since its signedness and its meaning are ambiguous

C++	kind	note
`std::string`	Text	`Mut<std::string>` has `set`, `insert`, `erase`, `replace`, `append`, each one splice
`std::array<T, N>`	Array	
`std::pair`, `std::tuple`	Struct	positional fields `"0"`, `"1"`, ... as Rust's tuples
`std::optional<T>`	Enum	`None` = 0, and `Some` = 1 holding the `T`, as `Option<T>`
`std::vector<T>` for fixed-size `T`	Sequence	element `i` lies at `i` times its size, so no index is needed
`kladde::Vector<T>`	Sequence	a `std::vector<T>` and a Fenwick tree of sizes, for variable-size `T`
`kladde::Map<K, V>`	Sequence of entries	the index map of the built-in containers; entries named as proposed below
`kladde::List<T>`	Sequence	elements with stable addresses, for long-lived references, below
`kladde::Box<T>`	as `T`	a non-null `std::unique_ptr<T>`; how a recursive type gets a size, as Rust's `Box`
`std::unique_ptr<T>`	as `std::optional<T>`	nullable, so an option
`kladde::Fixed<T>`	Fixed	or `[ [=kladde::fixed] ]` and the macro's option on a field
`kladde::Blob`	Opaque	bytes with a length, rewritten whole
`enum class`, `std::variant`, class hierarchies	Enum	below

A `std::vector` of variable-size elements is refused at compile time with a message that names `kladde::Vector`, rather than accepted with a hidden linear scan: it is the one place where the plain type would quietly cost what kladde exists to avoid.

`Mut<std::string>::set` splices only what changed: it finds the common prefix and suffix of the old and the new text, which costs one pass over memory the application has just written, and records one splice of the middle. Code that sets a whole string at every change, such as code that writes a path's whole `d` attribute for every step of a dragged control point, then records a few bytes rather than the whole attribute, without having to be rewritten in terms of edits. This belongs in every implementation's text type and is worth proposing for the built-in containers.

Decision: yes, state this as a guarantee in the spec for `Text` and `Blob`. More precisely, the guarantee should be that setting them to a new whole value should result in records that edit at most the smallest subrange that contains all changes (there's probably a more eloquent way to phrase this). Whether an implementation goes further and tries to detect if the changes are limited to multiple subranges that are far enough apart to justify more complicated optimizations is up to the implementation, but the spec should guarantee an optimization for the common case where some substring of a larger string (or `Blob`) has changed and the application author is simply too lazy to call the appropriate splice method and instead resets the whole string.

Claude: Agreed, with one refinement for blobs, and one question of where it goes. The guarantee holds as you state it for Text, which has no count: one splice of the smallest range that holds every changed byte. A blob is an Opaque without a fixed size, whose encoding starts with its length, so a blob whose length changes changes its first bytes as well, and the smallest range holding every change would then reach from the length to the last change — possibly the whole blob. So for anything with a length prefix the guarantee should be stated per part: the length is rewritten on its own, and the payload is spliced in the smallest range that holds its changes, which is two records in one transaction. The natural home

is the specification's guarantees an implementation must provide, beside the asymptotic complexity it already promises; I have not written it there yet, since the blob refinement is yours to accept.

@Claude: agreed, with one nit-pick: the phrasing shouldn't make it necessary to set the length on its own. If a blob changes at its start, an implementation may decide to write the length prefix and the changed content in one go.

25.6 Enums

The schema has one kind of enum, a discriminant followed by the fields of the variant it selects, and C++ has three ways to say it. All three map to the same descriptor kind, so which one a program uses is invisible in the file: one program may read with a `std::variant` what another wrote from a class hierarchy, as long as names and discriminants agree — a descriptor describes a representation, not an identity.

25.6.1 `enum class`: variants without fields

```
enum class Join : std::uint8_t { miter, round, bevel };
KLADDE_ENUM(Join, miter, round, bevel)
```

The discriminant is the enumerator's value, the variant's name its identifier, and the fixed form's discriminant width the underlying type's. The macro exists only to supply names; with reflection, `std::meta::enumerators_of` does, and the annotation `[ [=kladde::durable] ]` is all a reflected enum needs.

25.6.2 `std::variant`: a tagged union with value semantics

```
struct Mobile   { std::string digits; };
struct Landline { std::string digits; };
KLADDE_STRUCT(Mobile, digits)
KLADDE_STRUCT(Landline, digits)

using PhoneNumber = std::variant<Mobile, Landline>;
```

Each alternative is a variant, named after the alternative's described name, whose payload is the alternative — which presumes variants that hold one value, the first of the proposed changes to the specification. With it, one rule maps every Rust enum: kladde-svg's `Node`, which mixes a tuple variant holding a struct, `Element(Element)`, a tuple variant holding text, `Text(Str)`, and a struct variant, `ProcessingInstruction { target, value }`, is in C++

```
struct Text    { std::string content; };   // transparent: a Text is its string
struct Comment { std::string content; };
struct ProcessingInstruction { std::string target; std::string value; };
KLADDE_TRANSPARENT(Text, content)
KLADDE_TRANSPARENT(Comment, content)
KLADDE_STRUCT(ProcessingInstruction, target, value)

using Node = std::variant<Element, Text, Comment, ProcessingInstruction>;
```

with the same descriptor and the same fingerprint.

Without the change, every alternative needs a choice, and nothing in the C++ type can make it: today a Rust tuple variant `Element(Element)` holds one field `"0"` of type `Element`, while a struct variant `ProcessingInstruction { target, value }` holds two fields of its own, and the two descriptors differ although the encodings agree. kladde-cpp would then hold each alternative as a field `"0"` by default, which matches tuple variants and kladde-rs's enums in place of trait objects, and offer a flattening option per alternative for struct variants; `Node` would need it on `ProcessingInstruction` alone.

Discriminants default to the alternative's index, which is fragile in exactly the way Rust's implicit discriminants are: inserting an alternative renumbers the ones after it. `KLADDE_VARIANT(PhoneNumber, (Mo-`

bile, 1), (Landline, 2)) pins them, and the encoding-stability advice of kladde-rs carries over word for word.

Mut<std::variant<…>> offers set to switch alternatives, and visit, which calls a visitor with the handle of the current alternative, Mut<Mobile> or Mut<Landline>, as std::visit does with values; get_if<Mobile>() returns a handle or nothing.

25.6.3 Class hierarchies: a closed set of classes, held polymorphically

```
struct Shape {
    float x = 0, y = 0;
    virtual ~Shape() = default;
    virtual float area() const = 0;
};
struct Circle : Shape { float r = 1;    float area() const override; };
struct Rect   : Shape { float w, h;     float area() const override; };
struct Frame  : Rect  { float border;   float area() const override; };

KLADDE_STRUCT(Shape, x, y)
KLADDE_DERIVED(Circle, Shape, r)
KLADDE_DERIVED(Rect, Shape, w, h)
KLADDE_DERIVED(Frame, Rect, border)
KLADDE_HIERARCHY(Shape, (Circle, 1), (Rect, 2), (Frame, 3))

struct Drawing { kladde::Vector<kladde::Poly<Shape>> shapes; };
```

kladde::Poly<Base> is a non-null std::unique_ptr<Base> whose dynamic type must be one of the classes registered for Base, and it is durable as an enum with a variant per concrete class. In memory it is exactly what object-oriented C++ code holds already, so file.get().shapes[0]->area() is a virtual call and nothing more — no enum to forward through, and nothing like kladde-rs's #[kladde::dispatch], because C++ has the real thing.

- **Every concrete class is a variant, at any depth**: Frame derives from Rect, and both are variants, flat, of one enum; abstract classes are not.
- **A variant's payload is a struct of the members of its class and of its bases, bases first**, Frame's holding x, y, w, h, border, just as for an alternative of a std::variant.
- **Discriminants are explicit and belong to the hierarchy**, not to a field, so a field typed Poly<Rect> is an enum of Rect and Frame with the same discriminants, 2 and 3, as in a Poly<Shape>. Explicit, because classes are added to a hierarchy far more often than alternatives to a variant, and nothing orders them reliably.
- **The registry maps the dynamic type to its variant** through typeid, once per encode, so the classes need no virtual function of kladde's, and a class hierarchy that cannot be changed can still be made durable.
- **Decoding constructs the class**, by default construction, or through a factory registered with it for classes that have none, as Inkscape's objects do.

Mutating through a Mut<Poly<Shape>> needs no dispatch for the base's members: every variant's payload starts with them, right after the discriminant, so s.x().set(10) writes at the same offset whatever the shape is. For a derived class's members, s.as<Circle>() returns a Mut<Circle> if the shape is a circle, as dynamic_cast does for pointers, and s.set(std::make_unique<Rect>(…)) replaces the shape with another, by splicing the new encoding over the old.

25.6.4 Which to use

	enum class	std::variant	class hierarchy
fields per variant	none	any	any, inherited ones included

	`enum class`	`std::variant`	class hierarchy
in memory	the enumerator	inline, the size of the largest alternative	on the heap, the size of its own class
adding a variant	an enumerator	an alternative, and every `visit` must handle it	a class and a line in the registry
behaviour	free functions	`std::visit`	virtual functions
shared fields	—	repeated in each alternative	in the base, and writable without dispatch
suits	flags and choices	small closed sets of values, new code	existing object-oriented code, and open-ended sets of kinds

All three deserve first-class support, since they serve different code rather than different tastes. New code with a handful of small alternatives is better off with `std::variant`, which keeps values inline and visits them exhaustively; code that already has a hierarchy — Inkscape's `SPObject` and its dozens of subclasses — needs the hierarchy, and could not reasonably be rewritten into variants. The hierarchy is also the only form in which shared fields are written without dispatch, which a document model with many kinds of element and many common attributes uses constantly. And for mere flags, an `enum class` is often still the simplest choice.

25.7 Handles: where a value lies

A handle has to know where its value starts in the sequence whenever it writes, and has to tell its ancestors when its value changes size; Values and their positions gives the algorithms, and C++ needs both of its two forms.

25.7.1 Short-lived handles

`Mut<T>` is the guard of kladde-rs: made from its parent's handle by one lookup in the parent's index, holding its start, and valid while nothing outside its own subtree is written. Rust enforces that by borrowing the parent exclusively; C++ cannot, so it is a documented rule, the same kind as iterator invalidation, which C++ programmers already keep: *a handle is valid until something is written through a handle that is not its descendant.* Debug builds check it, with a stack of the live handles in the store that a write pops down to the handle it went through — the discipline that Miri's stacked borrows check for Rust — so a stale handle fails an assertion rather than writing at the wrong position. Release builds pay nothing.

Guards made together carry over unchanged: `parts()` hands out handles that compute their start at every write, so that writing one field and then another works whatever the first did to its size.

25.7.2 Long-lived references

An application that keeps pointers into its document for as long as it likes needs handles that survive edits elsewhere, and Rust's design deliberately has none. Inkscape holds `Inkscape::XML::Node*` everywhere — in the selection, in tools, in dialogs, in the undo log — and writes through any of them at any time.

`kladde::List<T>` keeps its elements at stable addresses, and each element knows its parent and where it stands among its siblings. A `kladde::Ref<T>` to an element is then valid until that element is removed, as an iterator of `std::list` is, and computes its start when it writes by climbing from the element to the root: at each level, the element's offset among its siblings, from a Fenwick tree over the siblings' sizes at the index the element keeps of itself, plus the offset of the list in its parent. @Claude: is there a specific reason you chose an order-statistics tree here over a Fenwick tree? Would a Fenwick tree not work here or does it have an unfavorable trade-off concerning memory or runtime?

Claude: No good reason: a Fenwick tree works, and it is the better default, so I changed the paragraph to it. I had reached for a tree because a `Ref` must find its element's rank from the element alone, and in

a tree with parent links that survives insertions before it in `O(log m)`. With a Fenwick tree, each element keeps its own index instead, and an insertion or removal renumbers the elements behind it and rebuilds the tree, both `O(m)` — the same `O(m)` that moving the elements' pointers in the list's array costs anyway, as the built-in containers argue for `kladde::Vector`. In return an element costs 4 bytes of the tree and 4 of its own index instead of a tree node of 24 to 40, and finding its rank is a load rather than a climb. An Inkscape element has tens to a few thousand siblings, for which renumbering costs microseconds; a tree would pay only for lists of hundreds of thousands that are edited in the middle, and can be an option for them, as it is for the vector.

That is `O(depth · log m)` per write, the cost of guards made together at every level, and a change of size updates the same path back up. It costs a parent link, an index and a cell of the siblings' Fenwick tree per element, about 16 bytes, which a document tree whose elements each hold a name, attributes and children affords.

So `kladde::Vector` and `kladde::List` differ the way `std::vector` and `std::list` do — compact, or stable — and an application chooses per container.

25.8 Errors, transactions and exception safety

@Claude: The statement in the specification that a C++ implementation of kladde should use exceptions was not based on any deep argument and shouldn't be taken as a law. I haven't been involved in any large C++ code bases for almost 20 years, so I'm not up to date about the state of exceptions in C++, neither in terms of technical changes nor in terms of community acceptance. I remember that some experts used to advise against the use of exceptions, or even against enabling exceptions in the compiler. Has the advice changed since? And is there a consensus in the community on how to do error handling in C++? What to popular code bases like chromium, llvm, react-native, llama.cpp (or others that might be more trend setting) use?

Claude: There is no consensus, and the split has barely moved in twenty years; what has changed is that the case against exceptions is now made for particular environments rather than for C++ as a whole. So kladde-cpp should work with and without them, and should use them only for failures that end a session.

Who disables them. Google's style guide still says "we do not use C++ exceptions", for the reason it always gave — its existing code is not exception-safe — and Chromium builds with `-fno-exceptions`. LLVM and Clang build without exceptions and without RTTI, and report errors through `llvm::Error` and `llvm::Expected<T>`, which abort if left unchecked in builds with LLVM's ABI-breaking checks enabled. Firefox, most game engines (Unreal disables them by default), and most embedded code do the same.

Who uses them. The standard library and Boost throw, and so do glibmm and gtkmm, which is why Inkscape builds with exceptions and catches `Glib::Error`. LibreOffice, KDE applications, Bloomberg's libraries and much scientific code use them, and the C++ Core Guidelines recommend them for "a function that cannot perform its assigned task". React Native's C++ core (JSI, Fabric, Folly) builds with exceptions; llama.cpp's C++ throws internally while loading a model and catches at its C API, while ggml, being C, returns codes and aborts on failed assertions.

What has changed. C++23 standardized `std::expected`, so value-based error handling no longer needs a homemade type, and it has become the usual choice for failures a caller expects, such as a file that does not parse. The cost argument has weakened: the happy path of an exception costs nothing on the common ABIs, and recent work on embedded targets (Khalil Estell's talks at CppCon 2024) found exceptions producing smaller binaries than result types. Herb Sutter's proposal of cheaper, deterministic exceptions (P0709) was not adopted, so the language itself is where it was. Widely used libraries cope with both camps by making throwing configurable — nlohmann/json and {fmt} both build without exceptions and abort or call a handler instead.

What that suggests for kladde-cpp. Its failures come in two kinds, which deserve different mechanisms:

- **Opening and creating a file can fail in ordinary ways** — the file is missing, damaged, or of another schema — and the caller always has to handle them, so `File<T>::open` returns an expected value (`std::expected` on C++23, a type of kladde's own on C++20), with or without exceptions.

- **A mutation or a flush fails only when the disk does**, and the session ends: the `File` is poisoned. With exceptions, the call throws; without, it poisons the `File`, calls a handler the application installs (Inkscape would show a dialog), and returns as if nothing had been recorded, which is true of the value, since memory changes only after recording succeeds. Making every `set` and `push_back` return a result that must be checked would put a branch after every edit in the application for a failure that, once it happens, makes all further edits moot.

The specification's "in C++ by throwing" should then say only that the implementation reports the failure and refuses further work.

Decision: I agree with your suggestion for error handling in kladde-cpp.

Errors are exceptions, as the specification already says of C++: a `kladde::Error` hierarchy, with a failed `fsync` poisoning the `File`, so that every later call throws until it is reopened. Applications built without exceptions are an open question.

Every mutation prepares, records, then commits, and only the first two may throw. Rust treats a failed allocation as fatal; C++ reports it, so a `push_back` that recorded its splice and then failed to grow the vector in memory would leave the file and the value disagreeing. So a mutation first does everything in memory that can fail without changing the value — reserving capacity, constructing the new element — then records, then changes memory by operations that cannot throw. This is the strong exception guarantee, earned in the order the reference algorithms already recommend.

Transactions take a callable, which is the one form that tells success from failure without asking a destructor to do I/O:

```
file.transaction([] (kladde::Mut<Accounts> a) {
    a.checking().set(70);
    a.savings().set(30);
});   // appended here, as one transaction
```

Returning commits; an exception leaves the callable with memory already changed and no way to change it back, so it poisons the `File` and propagates, as a panic does in kladde-rs. `file.batch(...)` has the same shape, and transactions nest inside batches as the state machine allows. An RAII object for transactions that span functions can come later; its destructor must not do I/O, so it would need an explicit `commit()` and would poison if destroyed without one.

25.9 Flushing on an interactive thread

A flush ends in an `fsync`, and a flush can be triggered from inside any mutating call, when the journal's budget would overflow. In Inkscape, mutating calls run on the GUI thread, so a drag could stall for as long as the disk takes to sync — milliseconds on an SSD, more on a slow disk.

Two remedies need nothing new:

- **Flush when the application is idle**: `file.flush()` from an idle callback, which an event loop offers, keeps the journal short, so the automatic flush rarely fires mid-gesture. kladde-cpp can offer the policy, `file.flush_if_journal_exceeds(fraction)`, and leave the scheduling to the application's loop. @Claude: explain: is the idea to add a method `flush_if_journal_exceeds` on `kladde::File<T>` that flushes if the journal is filled up to at least the given fraction? And does inkscape already have an idle loop where this method could be called?

  Claude: Yes to both. `file.flush_if_journal_exceeds(0.5)` appends a batch's ready run if there is one and flushes if the journal pages then hold at least half of their budget, and otherwise does nothing; the check is a subtraction of two cursors, so calling it often costs nothing. Inkscape runs GLib's main loop, and already does its own deferred work in idle callbacks: `SPDocument::requestModified` connects `SPDocument::idle_handler` with `Glib::signal_idle()` after every change, to update the objects built from the XML tree. The fork would connect a second idle callback the same way, at a priority below redrawing, so that it runs once events and frames are done — between gestures rather than during one. An idle flush ends the journal's epoch early, which matters for cuts that outlive their transaction as answered below, so the fraction should stay well above zero.

- **A generous budget**, so that a gesture fits in the journal however long it lasts.

A flush on a background thread is a different matter: mutations appended while it runs would belong to the next journal, and an application crash before the flush's header is written would lose them although their calls had returned. That needs a change to the journal, which is proposed below, and then removes the stall altogether.

Decision: go with the flush when the application is idle. Document the flush on the background thread as a possible improvement, pending experience with the demo app.

25.10 Files and platforms

The store reads and writes through a `Storage` interface, as kladde-rs does: a file, and an in-memory storage for tests that can simulate power cuts by dropping any subset of the writes since the last sync.

- **POSIX**: `pread` and `pwrite`, `fsync`, and `posix_fallocate` to grow the file, so that a full disk is reported before a flush rather than discovered at its `fsync`.
- **macOS**: `fsync` does not ask the drive to empty its cache, so the store calls `fcntl(fd, F_FULLFSYNC);` anything less breaks invariant I1 on power loss.
- **Windows**: `FlushFileBuffers`, and `SetFileValidData` or `SetEndOfFile` with care for growth.
- **No `mmap`**, for the reasons the reference algorithms give.

The store owns its file exclusively, and kladde-cpp locks it (`flock`, `LockFileEx`) so that a second process opening it fails cleanly rather than corrupting it.

25.11 The Inkscape demo

Inkscape funnels every change to a document through one abstract class, `Inkscape::XML::Node`, whose implementation is `SimpleNode`: `setAttributeImpl`, `setContent`, `appendChild`, `removeChild`, `changeOrder`, and a few more. The objects Inkscape draws (`SPObject` and its subclasses) are built from that tree and updated by observing it, and its undo log records and replays changes through the same interface. So a `KladdeNode` that implements `Node` over a durable tree makes every change durable without touching the tools, the canvas or the undo system.

The first schema should be XML's, not SVG's:

```
struct Attribute { std::string name; std::string value; };
struct Element;
using Node = std::variant<kladde::Box<Element>, Text, Comment, ProcessingInstruction>;
struct Element {
    std::string name;
    kladde::Vector<Attribute> attributes;   // in their order, as SimpleNode keeps them
    kladde::List<Node> children;            // stable, since Inkscape holds Node*
};
```

Every Inkscape operation is then one or two records: an attribute set is a splice of its value's changed middle, a reorder is a `Move`, an insertion a splice of the new subtree's encoding. Two details need care: Inkscape creates nodes detached and appends them later, so a detached node is an ordinary value until it is appended, which encodes it; and a node removed and appended elsewhere in one undo step can be a cut and paste rather than a second encoding, if the step is a transaction. A node removed in one step and inserted again in another — undoing a deletion is the common case — is encoded again, subtree and all, unless cuts outlive their transaction. Inkscape's undo steps (`DocumentUndo::done`) are natural batch boundaries.

The fork's schema is its own, and free to follow Inkscape's needs rather than any other program's.

A typed SVG schema is the showcase that should follow, with Inkscape's element classes as a hierarchy and attributes as typed fields, but it touches much more of Inkscape, and the XML schema already delivers the demo: no save button, and nothing lost to a crash. @Claude: I'd like to have this eventually, so that I can compare file sizes, memory usage, and run times in the XML-based and the typed schema.

But I agree that the XML-based schema should go first. Implement the typed SVG schema once you're done with the XML-based schema, and implement it on a new branch `typed` in `inkscape-kladde`.

Claude: Noted: the XML schema first, then the typed schema on a branch `typed` of inkscape-kladde, and the comparison of file sizes, memory and run times between the two.

25.12 Changes to the specification

Four changes would make kladde-cpp simpler or its API more idiomatic, and none of them is about C++ alone: each came up because a second language, or an application of a kind kladde-rs has not met yet, looks at the format from a different side. They are ordered by how much they would change kladde-cpp's design, and each says what it costs.

25.12.1 Variants that hold one value

Proposed: an enum's variant holds at most one value, its payload, of any type, instead of a list of fields. A fieldless variant has no payload; a Rust tuple variant of one field, `Element(Element)` or `Text(Str)`, holds that field's value; a struct variant, `ProcessingInstruction { target, value }`, holds a Struct of its fields, named after the variant; and a tuple variant of several fields a Struct of positional ones. A payload that is a Struct without fields counts as no payload, since both encode to nothing.

What it buys. Every C++ form of an enum — an alternative of a `std::variant`, a class of a hierarchy, `Some` of a `std::optional` — holds exactly one value, and with this rule that value is the payload, with no choice for the programmer to make and none to get wrong. Rust's two spellings of "a variant that holds a struct", `Element(Element)` and `Element { kind, attrs, … }`, get one descriptor, so the choice between them stops being a file-format decision, as a struct's own name already is not. And every language with tagged unions of one value — Swift's enums with associated values, TypeScript's discriminated unions, Python's union types — maps the same way.

What it costs. The canonical encoding of an Enum writes `reference(payload) + 1`, or `0` for none, in place of `varint(field_count)` and the fields; the fingerprint follows, and the vector for `List` changes, since `Cons` then holds a Struct of `"0"` and `"1"`. @Claude: I didn't understand the last part about the vector for `List`. Explain in more detail.

Claude: "Vector" there meant a test vector: the second row of the table of expected fingerprints, which the proposal would change. That row hashes this type graph:

```
[0] Enum "List", width 4 { 0 "Nil" {}, 1 "Cons" { "0": [1], "1": [0] } }
[1] i32
```

Under the proposal, `Cons` no longer holds two fields but one payload, a Struct of them, which becomes a descriptor of its own:

```
[0] Enum "List", width 4 { 0 "Nil": [2], 1 "Cons": [3] }
[1] i32
[2] Struct { }
[3] Struct { "0": [1], "1": [0] }
```

(`Nil` holds the empty Struct if variants always hold one value, as recommended below.) The graph is a different graph, so the hash input of `List` changes and the row's expected fingerprint, `af12af01…`, has to be recomputed; the bytes of every `List` value stay exactly as they were. No value's encoding changes, and the fixed form of an Enum is the payload's fixed form instead of its fields', which is the same bytes. kladde-rs's derive describes a struct variant as a Struct and a one-field tuple variant by its field's type. Files of today stop opening, which a pre-stable format allows.

@Claude: I like this proposal. One detail: I'm unsure whether variants should have *at most* one value or *always* one value (giving valueless variants a payload of type `()`). Briefly discuss the trade-off.

Claude: *Always* one value is the better rule: it makes every variant the same shape, and it removes the one special case that *at most* one needed.

- **What *always* buys.** A variant is a discriminant and a payload, with no "or nothing" in the descriptors, the fingerprint, the fixed form, or any code that walks them. And a fieldless variant, a

C++ alternative that is an empty struct, `std::monostate`, Rust's `()` and an `enum class` enumerator all get the same descriptor by construction — an empty Struct, whose name is not fingerprinted — where *at most* one needed a rule that "an empty Struct counts as no payload" to make them agree.
- **What it costs.** One descriptor in the table, the empty Struct, shared by every valueless variant, of three bytes; each variant then writes a reference to it, one byte, where *at most* one would have written a `0`, also one byte. The hash input of each valueless variant grows by a token of 17 bytes, which no one will measure.
- **What *at most* one would have bought**: an `enum class` described without an extra descriptor, and nothing else.

Decision: adopt this change, with the *always* one value refinement.

25.12.2 Maps as a kind of their own

Proposed: a descriptor kind Map, with a key type and a value type, whose encoding is that of a Sequence of entries, each its key's encoding and then its value's. Today a map is described as a Sequence of Structs of two fields, and the specification names no fields: kladde-rs calls them `"key"` and `"value"`, and a C++ `std::pair` would call them `"0"` and `"1"`, so the same map has two fingerprints until a convention is written down.

The least change is to write down kladde-rs's names. A kind of its own does more: every language has a native map, and a reader that knows a Sequence is a map can decode it into one — a `std::map` or a `kladde::Map` in C++, a `dict` in Python — and tools can render it as a JSON object rather than an array of pairs. It costs a kind tag, of which 132 is free, and nothing in any encoding.

25.12.3 A flush off the caller's thread

Proposed: a journal may end in a seal that names where the next epoch's journal begins, so that appends continue there while a flush folds the sealed one. A flush today blocks the mutating call that triggered it until its `fsync` returns, which is a stall on the GUI thread of an interactive application, mid-gesture. It cannot simply run beside new appends: those would go to the journal of the next epoch, which only the flush's header names, and an application crash before that header is written would lose them although their calls had returned — the guarantee that matters most.

With a seal, a flush starts by appending to journal `E`, as one more record, the page where journal `E + 1` begins, and then folds journal `E` up to the seal on a thread of its own, while the application appends to journal `E + 1`. Recovery from header `E − 1` replays journal `E`, reaches the seal, and continues into journal `E + 1` with its own epoch's salt; recovery from header `E` starts at the same page. So the transactions appended during the flush survive an application crash whichever header governs, and a power cut costs what it costs today. The seal needs no new encoding, only a meaning for the escape that a `Write` of no bytes reserves, and the pages of journal `E + 1` are live from the moment the seal reaches them, which the reuse rule then protects like any other. The flush's work moves to its thread, and the mutating call waits only when journal `E + 1` fills its budget before the flush ends.

This is not a matter of C++, but of the applications kladde-cpp is for: a document editor wants durability at every keystroke and a frame every few milliseconds. It should be measured in the Inkscape demo before it is adopted; idle-time flushing may turn out to be enough.

@Claude: would this proposal require making all of kladde thread save, i.e., add locks or synchronization primitives where necessary, and test for race conditions? If so, how intrusive would that change be? Or would a kladde implementation that can be reasonably used in inkscape have to be thread save anyway?

Claude: No to both: the application-facing API stays single-threaded, and only the flush hands work to a thread of its own, which with the seal can be as little as the `fsync` and the header write.

What Inkscape needs. Inkscape changes its document on the main thread only: the XML tree, the objects built from it and the undo log are all main-thread state. Its canvas renders on a thread pool, but from a snapshot of its own drawing tree (`Drawing::snapshot`, in `display/drawing.h`), not from the XML tree, so nothing else would touch kladde's values. So kladde-cpp, like kladde-rs, is used by one thread at a time, and its values, handles and containers need no locks.

What the seal needs. What takes long in a flush is the `fsync`; folding a journal is `O(r · log k)` in memory, and writing a flush's few dozen pages puts them in the page cache, both well under a millisecond for files of the target sizes. So the mutating call that triggers a flush can still seal, fold, place and write the pages itself, and hand only the `fsync` and the header write to a worker; the worker touches the file and nothing else. The shared state is then one flag, "a commit is in flight", with a condition variable: the next flush waits for it before taking any page, since the pages the previous commit freed become reusable only through its `fsync`, and the page pool learns of the commit when the worker reports it. Appending to journal `E + 1` while the worker syncs needs nothing at all: `pwrite` to other pages of the same file is safe beside an `fsync`.

How intrusive. One mutex, one condition variable, a worker thread, and a commit step that runs when the worker reports, all inside the store; the typed layer does not change. It is tested by ThreadSanitizer, and by the in-memory storage simulating a crash at every point of the interleaving, which the store's crash tests do already for one thread. Moving the fold and the writing to the worker as well would be more intrusive — the span table, the rope and the page table would then be the worker's during a flush, and journal appends would share the page pool with it — and is not needed while the `fsync` dominates.

Decision: implement the flush in the idle loop first. Make the change that flushes in a separate thread (and its change to the specification of the journal) in separate commits to kladde-docs, kladde-rs, and kladde-cpp. The spec should make it optional to actually use separate threads. Only the journal seal should be normative.

25.12.4 Cuts that outlive their transaction

Proposed: a removed value may be kept as a cut until the session ends, in a fourth chain that the header names, so that inserting it again, in any later transaction, is a paste rather than a second encoding. Values and their positions weighed this and deferred it "until a need is proven". A document editor with an undo system is that need: Inkscape keeps a deleted node alive for its undo log and inserts the same node again when the deletion is undone, by which time the transaction that removed it is long over. Without it, undoing the deletion of a large group writes all of the group's bytes again, into the journal and then into data pages.

It costs what the deferral listed: records that name a chain, the fourth chain in the header, and a rule for cuts nobody pastes — a splice in the cut chain when the in-memory value is destroyed, and the whole chain cleared by a session's first flush, since no value of an earlier session can name a cut. Whether it pays depends on how large the values are that Inkscape inserts again, which the demo can count before anything changes.

@Claude: help me think about the design of this in a bit more detail before I commit to a decison. My main objection is that I want a deletion of many items from a document or drawing to actually show up as a reduction in file size within a reasonably short time frame because I think that's what users expect. Therefore, keeping the whole undo buffer on file seems dangerous.

- Could we bound the capacity of this "side chain" by evicting old cuts? Could such a bound be implemented on the application layer (if we maybe give it some way to query encoding length of objects) or would it have to live in the store layer?
- Alternatively: instead of putting cuts on a durable side chain, could we turn them into some kind of "weak" pointers that stay alive as long as the corresponding spans still happen to be accessible somehow? For example, we could guarantee that cuts that haven't been pasted anywhere stay alive at least until the next flush. Flushes never happen within a transaction, so cut and paste within a transaction remains to always be possible, but across transactions they might require re-encoding. When pasting a cut, we would have to check somehow if the cut is still alive (a first implementation could make a conservative check and consider cuts to be still alive if they're from the current epoch, but a future implementation could also allow older cuts if their spans happen to still be alive). If a cut is still alive, it can be pasted cheaply; if not, it gets re-encoded. It's probably a bit more complicated than this since the transaction in which the `paste` record sits could trigger a flush before it gets written to a journal page, which might then invalidate the cut, and whether or not the `paste`'s transaction triggers a flush can't be known for sure at the time the `paste` record is issued.

Claude: Your second idea is the better design, and it settles the first: a cut should be a *weak* promise —

a paste is cheap while the cut's bytes are still at hand, and falls back to encoding the value otherwise — and it should live as long as the journal it was made in, rather than as long as its transaction. Nothing goes to file that would not be there anyway, so deleted content becomes garbage at the next flush, as it does today, and the file shrinks as consolidation reclaims it. The proposal above, a durable side chain, would not meet your requirement and should be replaced by this; I have left its text as it is until you decide.

Cuts that live as long as their journal. Three rules of the journal change, and nothing else in the format:

- **A `Paste` may name any cut that an earlier record of the same journal made and that is not pasted yet**, counted by distance among the journal's cuts, 0 for the latest; today it may name only cuts of its own transaction.
- **The fold keeps every cut until it ends**, rather than freeing a transaction's cuts when the next transaction starts, and replay at open does the same; both already hold every byte a paste can name, since it lies in the committed sequence or in the journal being folded.
- **A reader keeps the length of every cut of the journal**, not only of the open transaction, which the journal's page budget bounds.

The flush you worried about. A flush that lands between the cut and the paste does end the cut, and the way out is to decide the paste's form when its transaction is appended rather than when the application calls `paste`. Until then the buffer holds the operation as "paste cut `c`, or else encode this value", and the value is still the cut's own, since a cut gives no mutable access to it. When the transaction closes, the flush triggers run with the paste's size; if the transaction fits, it is appended with the `Paste`, into the journal that made the cut; if a flush must come first, the cut's journal is folded, so the paste is encoded as a `Splice` of the value's bytes instead, the transaction's size is computed again, and the rules continue from there, an overlong transaction included. Two cases need no fallback at all: a cut and its paste in one transaction, which no flush can separate, as today; and the ready run of a batch, which is appended before the flush that cuts the batch. With the seal, the rule is the same: a paste is a `Paste` only if it is appended to the journal that made its cut.

What the application sees. Nothing but cheaper inserts: Inkscape's removed node remembers its cut, and inserting it again asks the store whether the cut is still pasteable, which is one comparison of epochs and one bit. The application never needs an encoded length; the fallback encodes from the in-memory value, which Inkscape keeps anyway for its undo log. How long a cut lives is then how long a journal lives, so idle flushes shorten it; an undo right after a deletion is cheap, an undo after the next flush encodes again.

Cuts that outlive a flush, if measurements ask for them. A cut can survive a flush only if its bytes stay reachable from the header, which is the side chain again; so the side chain is not an alternative to weak cuts but their extension. Bounding it would then be the store's job, since only the store knows which bytes each cut holds and only a store record can drop them: it would evict the oldest cuts at a flush once the chain exceeds a byte budget, and an evicted cut would fall back to encoding like any other dead one. The application would need no lengths for this either, only the budget as an option. Whether it is worth a fourth chain depends on how often Inkscape inserts a node again after a flush, which the demo can count once weak cuts exist.

Decision: go with your formulation of "Cuts that live as long as their journal".

25.12.5 Smaller points

- **The rationale for SHA-256 says that it is "present in every mainstream language's standard library"**, which is not true of C or C++. That is no reason to change the hash — SHA-256 is a few hundred lines, and in C++ it can run in a `constexpr` function, so that a type's fingerprint is a compile-time constant and the fast path compares two constants — but the sentence should say so.
- **No change is needed for class hierarchies.** Flattening a class and its bases into one struct, bases first, is a convention of kladde-cpp, and a descriptor for "a struct that extends another" would describe the C++ source rather than the bytes, against the guiding principle.
- **No change is needed for C++ strings either**, although `std::string` holds bytes of any kind and the specification requires UTF-8: a handle validates the bytes it inserts, which costs a pass over what it writes anyway. @Claude: we could relax the rules about string encoding a bit if this helps. The one rule we need is that kladde uses `0xFF` as the string terminator, so a C++ implementation

has to deny strings that contain `0xFF`. How realistic is it for `0xFF` to appear in a string in an app that uses either a different string encoding or that claims to use utf8 but doesn't enforce it as strictly as rust?

Claude: In text that really is UTF-8, `0xFF` never appears, and in text that is not, it appears often enough that a relaxed rule would not make such strings safe to store — so the rule is best kept as it is.

- **Other encodings**: in Latin-1 and Windows-1252, `0xFF` is "ÿ", rare but real (in French place names, say); in UTF-16 and UTF-32 it is in the byte-order mark `FF FE` and in every code unit of "ÿ" or of the fullwidth forms `U+FF01…`; an application keeping such text in a `std::string` meets it routinely.
- **`std::string` as a byte buffer**, for file contents, serialized messages or binary identifiers, which is common C++: any byte occurs.
- **"UTF-8, but not enforced"**: the bytes come from outside — a file declared UTF-8 that is really Windows-1252, a Linux file name, the clipboard of another program — and then `0xFF` appears exactly as often as the foreign encoding produces it.
- **Inkscape**: its XML goes through libxml2, which transcodes documents to UTF-8 and rejects invalid sequences, and GLib's strings are UTF-8, so in practice never.

Relaxing Text to "any bytes but `0xFF`" would save C++ little, since validating the bytes a handle inserts runs at several gigabytes a second (the simdutf library, or a plain loop for short edits), and it would cost every other reader: Rust's `String`, Python's `str` and a tool's JSON export all need UTF-8, and would have to validate at open or fail there. Applications with genuinely binary or foreign-encoded strings are better served by a byte type that says so — a Sequence of `u8`, or a Bytes kind if editing them in place should cost one splice as text does.

Decision: I agree, `Text` should stay to require UTF-8. The API documentation of kladde-cpp should warn users that they must use `std::string` in kladde only for UTF-8 text, not as a byte buffer, and should point them to the appropriate type to use for byte buffers in kladde (presumably `std::vector<std::uint8_t>`).

25.13 Building, packaging and testing

- **CMake and C++20**, no required dependencies: CRC-32C and SHA-256 are a few hundred lines in the tree, with the hardware CRC instruction where the platform has one. A target `kladde::kladde` for `find_package` and `FetchContent`; packages for vcpkg and Conan once the API settles.
- **One namespace, `kladde`, and headers under `<kladde/…>`**: `kladde/store.hpp` for the untyped layer, `kladde/kladde.hpp` for everything.
- **Tests**: the specification's fingerprint vectors first; the store against a byte vector, with flushes, reopens and simulated power cuts; values against their canonical encoding after every operation; and the crash tests of Conformance. GoogleTest, which Inkscape uses already, and libFuzzer harnesses for every decoder, under AddressSanitizer and UndefinedBehaviorSanitizer.
- **Cross-implementation tests from the start**: kladde-cpp and kladde-rs each write a corpus that the other reads, compared through a textual dump of the schema and the sequence, the canonical textual representation in embryo. This is the first time the compatibility contract is tested by two implementations, and the most valuable thing kladde-cpp can do for the specification.

25.14 Order of work

1. **The schema**: descriptors, canonical encoding and fingerprints, checked against the specification's vectors and kladde-rs — after deciding on the two proposed changes to descriptors, so that neither implementation writes this layer twice.
2. **The store**, without consolidation: storage, pages, the span table, the rope, the journal, the flush, opening; the differential oracle and the crash tests.
3. **Typed values with the macro**: primitives, standard types, `kladde::Vector`, `kladde::Map`, structs, `enum class`, `std::variant`, `File`, `Mut`, transactions and batches.
4. **Cross-implementation tests** against kladde-rs.
5. **`kladde::List` and `Ref`**, and the Inkscape `KladdeNode` with the XML schema: the demo.

6. **Consolidation**, the header as write buffer and the span policy, ported from the reference algorithms.
7. **Class hierarchies**, and a typed SVG schema.
8. **Reflection**, once a compiler that the target applications build with supports it.

Consolidation comes late on purpose: a store that never consolidates is conforming, and only its files grow, which a demo can live with for a while.

25.15 Open questions

- **The four changes to the specification**: which to adopt now, which once the Inkscape demo has measured the need, and in what order relative to kladde-cpp's own work. Variants that hold one value and maps as a kind of their own are cheapest before kladde-cpp's schema layer is written, since both change descriptors and fingerprints.
- **Long-lived references in the reference algorithms.** Values and their positions describes handles that live as long as a guard; `kladde::List` adds a form that lives as long as its element. Is it a C++ matter, or does every language without a borrow checker — Python, Java — need the same, so that it belongs in Implementation? **Decision:** I think it belongs in Implementation. **Claude:** Agreed; it would go into Values and their positions as a second kind of handle beside the guard, once the design above has settled, with the Fenwick tree and the per-element index as its default.
- **The minimal splice of `set`** for text, and perhaps for any value whose encoding is long and changes little: worth stating in the built-in containers?
- **Builds without exceptions.** Some applications and platforms disable them; a parallel API returning `std::expected` would double the surface, and aborting on error would serve no one well.
- **Debug checking of handles**: is a stack of live handles cheap enough to keep on in the test builds of applications, or only in kladde-cpp's own?
- **The name of the handle**: `Mut<T>` is short and says what it is for, but reads oddly to C++ programmers who have not met Rust; `Edit<T>` or `Handle<T>` are the alternatives. @Claude: the name `Guard` in kladde-rs is historic. The name `Mut` was considered as an alternative when guards were first introduced, but discarded. I don't remember anymore whether the arguments against `Mut` were based on parts of the kladde design that have since gone. Would it make sense to rename `Guard` to `Mut` in kladde-rs at this point? It might lead to some confusion given that I think guards hold a shared reference (i.e., *not* a `&mut`) to the backend or store via a `RefCell` (correct me if I'm wrong here). What are the arguments for and against renaming `Guard` to `Mut` in kladde-rs?

  Claude: Your picture of kladde-rs is right, and I would rename, but to per-type names ending in `Mut`, as Rust's standard library names such wrappers, rather than to one generic `Mut`. A guard holds `&'r mut T`, the value, and a backend that is `Copy` — a shared `&Store` — whose state sits in a `RefCell<Inner>`.

  - **For.** The standard library calls a wrapper that grants exclusive access to a value and does something on the side `RefMut` or `PeekMut`, and reserves `Guard` for RAII objects whose drop releases something, as `MutexGuard` unlocks. A kladde guard's drop does nothing, so `Guard` promises a release that never happens. What a guard is exclusive about is the value, which it does hold by `&mut`; the shared reference to the store is invisible in its API, and a `RefMut` holds a shared reference to its `RefCell` in just the same way, so it should not confuse anyone. And kladde already names its accessors `entries_mut()`, which would then return a `DurableVecMut`; `Kladde::guard()` would become `Kladde::get_mut()`, beside `get()`; and kladde-rs and kladde-cpp would share one word.
  - **Against.** The rename touches everything that names guards: the trait, every derived `…Guard` type, the tutorial, and Values and their positions, which uses "guard" as the language-neutral word and would need another one, "handle" being the obvious choice. And a trait called `Mut` alone would say little, which is why the per-type names matter.

  Nothing I can find in the documents explains why `Mut` was discarded before, so I cannot tell whether the reason still holds.

Decision: rename `TGuard` to `TMut` (and associated types `Guard` to `Mut`, etc.) in kladde-rs and in kladde-docs. Use the name `Mut` in kladde-cpp too, even if it is foreign to C++. It's short, pretty self-explanatory, and avoids a needless deviation between the rust and C++ API.

Chapter 26

Evaluation

Measurements of kladde-rs and, in the Inkscape demo, kladde-cpp under realistic workloads, and what they say about the design.

Every page but the last measured the design that the present one replaced, of allocations and an address table: "main" in them is kladde-rs's main branch as it stood then, which holds that design and is kept as the branch `address-table`. Their findings about consolidation's policies — the churn floor, the budget, free filling, compaction mode — carry over to pages of spans, since the present design keeps those policies; their findings about statements, allocations and layouts do not. The last page, the file as one sequence, measured, compares the present design with that main.

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. Benchmarks and tests describes the programs behind them, and the tests of kladde-rs: what each runs and records, how to run it, and how to read what it writes.

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.

26.1 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

page	what it measures
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
Cleaning edited drawings and texts	main's churn floor and the three ripeness estimates, under the newest size rule, on three edited SVG drawings in three layouts and on two typed texts, cleaning until the live pages are about nine tenths full: live pages, data pages and leaves, their fill, which policy lies lowest, the files' free pages and closed sizes, writes, free filling, and times
Ripeness with main's free filling	the three ripeness branches of the page above with main's free filling in place of the draft's: how often free filling cleans, the pages the branches open and the bytes they write for it, how far each branch's curve moves toward the churn floor's, and the live pages, fill, files and writes at each target
Porting kladde to C++ and into Inkscape	what a second implementation, kladde-cpp, and the Inkscape demo found: compatibility of the two implementations' files and types, what the specification left open, what C++ made harder than Rust, what Inkscape needed, and the bugs the port found
The file as one sequence, measured	the present design, implemented on a branch of kladde-rs while it was a proposal, on the drawings and texts of the cleaning page under the churn floor, against main in its three layouts: the trade-off between space and writes, the encoding's size, live pages and their fill, the span policy's bound, files, writes, and times

Chapter 27

Benchmarks and tests

Every program that tests kladde-rs or measures it, in kladde-rs and in this repository: what each runs, what it records, how to run it, and how to read what it prints. The results are on the other pages of Evaluation; this page has none.

It describes kladde-rs at commit `7c5cdd6` on its branch `text-bench`, which is the branch `defrag-shifts` and one more benchmark, and which holds the design of allocations and an address table that every evaluation page but the last measured; the branch `address-table`, kladde-rs's main branch at the time, lacks `text-bench` and most of what kladde-svg measures.

kladde-rs's `main` now holds the present design, and keeps less of this. It has `text-bench`, and `svg-bench` with one layout, since every list and string is inline in one sequence, with the options `--quick`, `--edits`, `--only`, `--seed`, `--variant`, `--focus`, `--focus-moves`, `--journal-budget`, `--churn-floor` and `--target-fill`, and the span policy's `--span-min` and `--rewrite-below`; the file as one sequence, measured describes its tables. `kladde-bench`'s synthetic scenarios, which allocated, resized and freed, went with the design they measured. Its tests are the store's oracle, against a byte vector, its simulated power cuts and its consolidation, the containers' randomized tests, the derive's, and kladde-svg's round trip.

27.1 At a glance

program	where	what it does	what it leaves
`cargo test --workspace`	kladde-rs	unit, integration and documentation tests, among them a differential oracle and simulated power cuts	pass or fail
`kladde-bench`	kladde-rs, `crates/kladde-bench`	six synthetic workloads on the store, and variants of the options	one table per workload
`text-bench`	kladde-rs, `crates/kladde-bench`	typing into one long text	one table, and a summary
`svg-bench`	kladde-rs, `crates/kladde-svg`	a drawing editor's edits on an SVG drawing	one table, and a summary
`svg-roundtrip`	kladde-rs, `crates/kladde-svg`	every SVG file of a corpus through kladde and back, and their sizes	one table, a summary, and an exit status
`plot-evaluation.py`	kladde-docs, `tools/`	the figures of the evaluation pages, from the tables	SVG figures, and a summary table

program	where	what it does	what it leaves
`simulate-ripeness.py`	kladde-docs, `tools/`	a simulation of the ripeness drafts' estimators	tables of their costs
`check-examples.py`	kladde-docs, `tools/`	the Rust examples of these pages, compiled and run against kladde-rs	pass or fail
`make-pdf.py --check-only`, `check-site-links.py`	kladde-docs, `tools/`	this repository's links and page list	pass or fail

kladde-rs runs no continuous integration; this repository's runs on every push.

Every benchmark is deterministic but for its times: the same commit, options and seed make the same operations and write the same tables. Each one checks, at the end, that the reopened file reads what the program holds in memory, and panics otherwise, so a table from a run that finished describes a file that was right.

27.2 The tests

```
cargo test --workspace
```

The suite takes about half a minute once built, in debug mode, where the store's debug assertions check its invariants as it goes: among others, that a page a flush retires holds no live statement. `Store::check`, which the tests call after flushes, cross-checks the fragment map, the statement slab, the pins and every page's coverage, and panics on the first disagreement. Building with `--release` skips the assertions, so the tests are meant for debug builds.

crate	tests	what they cover
`kladde-store`	`tests/oracle.rs`	the differential oracle: allocations, frees, resizes, writes, splices, copies and moves, chosen at random, against a model of byte vectors; a check of every allocation's size and bytes and of `Store::check` after one step in twenty, and a reopening from the storage's bytes after one in a hundred; small and large allocations, with and without consolidation, and with a journal of one page
	`tests/crash.rs`	power cuts: `MemoryStorage::with_crash_tracking` fails the storage after a random number of operations, and the file as it would be with none of the writes since the last sync, all of them, or four random subsets of them, must recover onto a transaction boundary no earlier than the last completed flush

crate	tests	what they cover
	`tests/consolidation.rs`	that consolidation keeps a file near its live size under overwrites, appends and a mass free, without changing what it reads
	`tests/churn.rs`, `tests/restate.rs`, `tests/sizes.rs`	id recycling, bytes that splices shift within an allocation, and the newest size rule, each checked in a reopened file
	unit tests	the CRC, the fold, loading, pages, the journal, and the encoding of statements
`kladde-types`	`tests/packed.rs`, `tests/small.rs`	packed vectors and small values under random edits through every mutating method they offer, against a model after every edit and a reopened file after every few
	the other `tests/`	the whole stack through `Kladde`, enums' `parts()`, `move_element`, schemas, tuples
`kladde-derive`	`tests/`, and `compile_fail` examples	what `#[derive(Persistable)]` generates, and what it must refuse to compile
`kladde-schema`, `kladde-persist`, `kladde-varint`, `kladde`	unit tests	descriptors, canonical encoding and fingerprints; the serialization layer; varints; the facade
`kladde-svg`	`tests/roundtrip.rs`	the canonical form of SVG, what the model types, and a drawing's trip through kladde and a reopening, byte for byte
every crate	documentation examples	that every public item's example compiles and runs

The randomized tests are seeded, so a failure repeats. They run a range of seeds each; the crash test names the failing seed in its message, and for the oracle, narrowing the range finds it. A seed that found a defect is kept as a test of its own, as `a_grow_witness_in_a_reused_page_survives` keeps seed 229.

Some tests run only with a feature of kladde-svg, which the workspace's run leaves off:

```
cargo test -p kladde-svg --features f64      # decimals as f64
cargo test -p kladde-svg --features packed   # and small-ids, small-attrs, small: the layouts
```

27.3 The tables

`kladde-bench`, `text-bench` and `svg-bench` write the same table: a CSV file with one row per explicit flush, which `plot-evaluation.py` reads whichever program wrote it. A run first builds its file, which is not measured, then records a row after every interval of operations; the columns are the run's identity, a snapshot of the file, the work done since the measured phase began, and two times.

column	meaning
`scenario`	the workload: a `kladde-bench` scenario, `text`, or `svg`
`variant`	what distinguishes runs of one workload: `on` or `off` for consolidation, the name of a variant of the options, the text's size, or the drawing
`size`	the workload's size in bytes: its nominal live size for `kladde-bench`, the text's size at the start for `text-bench`, and the drawing's live bytes once stored for `svg-bench`
`flush`	rows so far
`ops`	operations since the measured phase began: writes, allocations and the like for `kladde-bench`, keystrokes, or edits
`app_bytes`	the bytes the application wrote since then, by the program's own measure (see each program)
`file_pages`	the file's length, in pages of 4,096 bytes
`data_pages`, `table_pages`	live data pages, and live address-table pages without the headers
`free_pages`	pages within the file that hold nothing live: reusable, or retired by the last commit
`live_data`, `live_table`	the bytes of those data and table pages that current state relies on
`alloc_bytes`, `allocations`	the sizes of the live allocations, summed, and their number, without the store's own consolidator state
`statements`, `fragments`	live statements in the address table, and entries in the fragment map
`budget`	the consolidation budget, in pages, as the controller set it
`data_written`, `table_written`, `headers_written`	pages written since the measured phase began: data pages, table pages without the headers, and headers
`journal_bytes`	bytes appended to the journal since then
`fresh_bytes`	bytes of the application's content that flushes wrote into data pages
`evacuated_pages`, `evacuated_bytes`	data pages that consolidation evacuated, and the live bytes it wrote again to do so
`free_filled`	victims taken into pages the flush wrote anyway
`budget_pages`	data pages opened only to take victims
`table_rewrites`	table pages that a page rewrite drained
`window_restated`	fragments that the rotating window restated
`defrag_rewrites`, `defrag_bytes`	rewrites by description defragmentation, and the bytes they wrote
`compaction_flushes`, `truncations`	flushes in compaction mode, and truncations of the file
`flush_us`	the explicit flush that ended the interval, in microseconds, its `fsync` included
`ops_us`	the interval's operations, in microseconds: for `kladde-bench` the whole loop, choosing the operations and keeping the model included; for `text-bench` and `svg-bench` the time in kladde's calls alone

column	meaning
`restated_bytes`, `merged_bytes`	bytes that splices and moves shifted within their allocation, which flushes stated at their new offsets where they lie, or wrote again to merge the fragments they lay in

27.3.1 Reading a table

- **The counters of work are cumulative**: the difference between two rows is the work of the interval between them, and the last row's is the run's.
- **The bytes written** are `journal_bytes` plus 4,096 bytes for every page in `data_written`, `table_written` and `headers_written`. Over `app_bytes`, they are the *write amplification*; over `ops`, the bytes per operation.
- **The file's size over its live size**, `file_pages` × 4,096 over `alloc_bytes`, is the *space amplification*; the live pages, (`data_pages` + `table_pages`) × 4,096 over `alloc_bytes`, leave out the free pages, the headers and the journal.
- **A page's fill** is its live bytes over the 4,081 bytes of content it holds, so `live_data` over 4,081 × `data_pages` for the data pages, and likewise for the table.
- **The steady state** is what the evaluation pages average over the second half of a run, after the file has settled from its fill; its first rows still show the file the fill left.
- **Times** depend on the machine and vary between identical runs, so they compare only between runs on one machine, and best by their medians over rows; ratios such as bytes per operation carry over to other machines. The evaluation pages name the machine and how far identical runs differed on it.

27.4 `kladde-bench`: synthetic workloads

```
cargo run --release -p kladde-bench -- <output directory> [scenario ...] [quick]
```

Each scenario fills a file, then writes to it until the application has written eight times its live size, flushing after every 1000 operations, and records a row per flush in `<scenario>.csv`. Without a scenario, all of them run, which takes a few minutes; `quick` makes every one small. The files go to `files/` in the output directory and are removed when the run is done.

scenario	live size	what the application does	`app_bytes` counts	variants, one run each
`uniform`	1, 8 and 64 MiB	writes of 1 to 511 bytes at random offsets of random allocations of 16 KiB	the bytes written	consolidation `on` at each size, and `off` at 1 and 8 MiB
`skewed`	as above	the same, with nine writes in ten going to a tenth of the allocations	as above	as above
`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	the records	`on`, `off`

scenario	live size	what the application does	`app_bytes` counts	variants, one run each
`churn`	16 MiB	values of 16 to 128 bytes, or one in five of 1 to 8 KiB, replaced by a new allocation, rewritten whole, or patched in 1 to 64 bytes	the bytes written	`on`, `off`
`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	the patches	`on`, `no-compaction`, `off`
`typed`	16 MiB	a `Kladde<PersistableHashMap<u64, PersistableString>>` of strings of 50 to 500 bytes, through the typed layers: 40 % replace a string, 30 % append to one, 20 % insert one, 10 % remove one	the strings' bytes	`on`
`tuning`	8 and 16 MiB	`uniform` and `skewed` at 8 MiB with a churn floor of 0.5 or 2, or a target fill of 0.6 or 0.5; `append` with 4 and 16 pages of defragmentation share	as above	one table per workload, `tuning-<workload>.csv`, with the default among the variants

With `quick`, every live size is 1 MiB, and `shrink`'s 4 MiB. The `shrink` scenario flushes every 100 patches rather than every 1000 operations, and its first row follows the mass free.

The branches of the cleaning policies extend it, and their evaluation pages say so where they use it: the ripeness branches add a scenario `mixed`, skewed overwrites of allocations that are each hot, cool or cold, and five columns after the others, `kappa`, the price of space their controller sets, and `mixed_pages`, `mixed_bytes`, `cold_bytes` and `cold_mixed_bytes`, how the three classes share data pages. On those branches, `KLADDE_BENCH_NO_STATE` in the environment makes the stores keep no consolidator state, and `KLADDE_BENCH_MYOPIC` makes them rank pages by the myopic rule.

27.5 `text-bench`: typing into one text

```
cargo run --release -p kladde-bench --bin text-bench -- <output directory> \
    [--quick] [--size BYTES] [--keystrokes N] [--seed N] [--variant NAME] \
    [--focus SHARE] [--focus-moves N] \
    [--journal-budget PAGES] [--shift-decay FACTOR] [--churn-floor λ] [--target-fill τ]
```

A text of random words in one allocation of the store is edited keystroke by keystroke at a cursor, as a text editor would; every keystroke that changes it is a splice, which makes this the benchmark of what splices cost. The text is 64 KiB by default, words of one to nine lowercase letters each followed by a space, and is stored below the typed layers, so that only the store is measured. Each of the 100,000 keystrokes, 10,000 with `--quick`:

share	keystroke
85 %	type a letter at the cursor, which moves past it
10 %	delete the byte before the cursor, if there is one
4.5 %	move the cursor
0.5 %	paste 200 random letters at the cursor, which moves past them

Nine moves in ten land in a window of a tenth of the text, which jumps to a random place every 5000 keystrokes, and the tenth anywhere; `--focus` sets the window's share, `--focus-moves` the keystrokes between its jumps, and `--focus 0` sends every move anywhere. The text grows with the typing, to about 3.7 times its size at the defaults.

Every 1000 keystrokes the store is flushed and a row goes to `text-<variant>.csv`, where the variant is the text's size in KiB, as in `64k`, unless `--variant` names it. `app_bytes` counts the bytes typed and pasted; deletions count nothing. `--journal-budget` sets `Options::journal_budget_pages`, so that the store also flushes by itself whenever a transaction would not fit in that many journal pages; `--shift-decay`, `--churn-floor` and `--target-fill` set the options of those names.

The summary on stderr has five lines:

1. the keystrokes of each kind, the text's size before and after, and the flushes, of them those at rows;
2. what kladde wrote, per keystroke, split into the journal, data pages, table pages and headers;
3. the bytes per keystroke that splices shifted, restated where they lie and written again;
4. the statements, fragments, table pages and data pages the file holds at the end, and the median flush;
5. the closed file's size, over the text's, and the median of 21 reopenings.

27.6 `svg-bench`: a drawing editor

```
crates/kladde-svg/corpus/fetch.sh
cargo run --release -p kladde-svg --bin svg-bench [--features LAYOUT] -- \
    <drawing.svg> <output directory> [--quick] [--edits N] [--only KIND] [--seed N] [--variant NAME] \
    [--focus SHARE] [--focus-moves N] \
    [--journal-budget PAGES] [--shift-decay FACTOR] [--churn-floor λ] [--target-fill τ]
```

kladde-svg stores an SVG drawing in a model of `#[derive(Persistable)]` types, and `svg-bench` edits it with a seeded mix of what a drawing editor does. `fetch.sh` downloads the corpus, pinned by SHA-256 and not committed: the Ghostscript tiger, the coat of arms of Spain and a blank world map from Wikimedia Commons, under `drawings/`, and resvg's test suite at v0.48.1, under `resvg/`.

share	edit	what it does to the model
30 %	`Drag`	set the translation that leads a shape's transform list, or insert one
20 %	`MoveNode`	set the end point of a segment of a path's data
8 % / 7 %	`InsertNode` / `DeleteNode`	insert a relative line into a path's data, or delete a segment other than the first
10 %	`Recolor`	set a shape's fill colour where it is set, in its attributes or its style, or add a fill attribute
8 % / 7 %	`Duplicate` / `Delete`	insert a deep copy of a shape right above it, or delete one; each turns into the other when the drawing has grown or shrunk by a fifth
5 %	`Restack`	move a shape elsewhere among its siblings, with `move_element`

share	edit	what it does to the model
5 %	`Text`	type into a text, or replace it once it is long; a recolour in a drawing without text

The edits keep to one part of the drawing at a time: nine in ten pick from a window of a tenth of what they pick from, the shapes, the paths, the shapes without children or the texts, and the window jumps every 3000 edits. `--focus` and `--focus-moves` set the window as in `text-bench`; `--focus 0` picks every edit from the whole drawing, except that nine drags in ten move one of the first tenth of the shapes, which was the benchmark's only mix before kladde-rs `e88fd47`. `--only KIND` makes every edit one kind.

A run ends once the edits have stored eight times the drawing's live size, once with `--quick`, or after exactly `N` edits with `--edits`, which is how runs of different layouts, branches or options make the same edits: the evaluation pages give each drawing's number. Every 1000 edits the drawing is flushed and a row goes to `svg-<variant>.csv`, where the variant is the drawing's file name unless `--variant` names it. `app_bytes` counts what each edit stores: the bytes of the numbers it sets, or the encoding of the value it stores, in its place, with everything that value owns; deletions count nothing. The options of the store are set as in `text-bench`.

The layouts are cargo features of kladde-svg, each adding to the one before: none, the *slotted* layout of fixed slots; `packed`, lists of enums packed; `small-ids`, ids as small strings; `small-attrs`, attribute lists and strings small; `small`, every list small. `f64` stores decimals as `f64` instead of `f32`. Each layout is a build of its own, so a script that compares them builds once per feature and copies the binary aside.

The summary on stderr gives the drawing's size in SVG and in kladde and what it holds to edit; the edits, the bytes they stored and the mix as it came out; what kladde wrote, per edit and per byte stored, and the file over its live size; what saving the whole canonical SVG every 1, 100 or 1000 edits would have written instead; the edits per second in kladde's calls and the median flush; the closed file's size; and the times to parse the SVG, to reopen the closed file, as the median of 21, and to write the SVG. After the edits, the reopened drawing must write the same canonical SVG as before closing.

27.7 `svg-roundtrip`: a corpus through kladde

```
cargo run --release -p kladde-svg --bin svg-roundtrip [--features LAYOUT,render] -- \
    [--csv <table.csv>] [--render] [--journal-budget PAGES] <file or directory> ...
```

For every `.svg` file under the paths, it checks that the canonical form is stable and that kladde gives it back byte for byte, and records the sizes of the file in each form. The canonical form is what kladde-svg writes: no whitespace between tags outside text, single spaces between attributes, and path data in the grammar's shortcuts. It exits with a failure if any canonical form was not idempotent or any trip through kladde differed.

column	meaning
`file`, `status`	the file, and `ok`, `not-idempotent`, `kladde-mismatch`, `not-xml`, `not-utf8` or `unreadable`
`orig_bytes`, `orig_gz`, `canon_bytes`, `canon_gz`	the original and the canonical form, plain and gzipped at the highest level
`kladde_bytes`	the kladde file after one flush and `close`
`settled_bytes`	the same file once two more sessions have opened and closed it, which returns the free pages a new file holds ahead of its data
`alloc_bytes`, `allocations`, `statements`	the store's live allocation bytes, allocations and statements
`data_pages`, `table_pages`, `free_pages`	the first-closed file's pages by kind

column	meaning
`numbers`, `numbers_changed`	the numbers stored, and those whose shortest decimal changed by being stored as the model's number type
`attrs`, `other_attrs`, `attr_bytes`, `other_attr_bytes`	attributes, and those the model kept verbatim, by count and bytes
`idempotent`, `kladde_ok`	the two checks, as 1 or 0
`render_max_diff`, `render_diff_share`	with `--render`: the largest difference of any channel of any pixel between the original and the canonical form, rendered by resvg, and the share of pixels that differ by more than 1; a difference is a gap in the model, not a defect of kladde
`parse_us`, `open_us`	the time to parse the SVG, and to reopen the kladde file

The summary on stderr counts the files by status, names those that failed, and sums the sizes.

`svg2kladde <in.svg> <out.kladde>` and `kladde2svg [--indent] <in.kladde> <out.svg>`, binaries of the same crate, convert single files, for looking at one by hand.

27.8 Tools in this repository

27.8.1 The figures

```
tools/plot-evaluation.py [--only NAMES] [--summary] --out <directory> <label>=<directory> ...
```

Draws the evaluation pages' figures from tables, gzipped or not, each directory a run and each label its name in the figures. It recognizes the tables by name: `kladde-bench`'s by their scenario, `svg-bench`'s as `svg-<drawing>.csv`, and `text-bench`'s as `text-<size>.csv`. `--only` draws the figures it names, out of `space-amplification`, `mixed`, `live-fraction`, `write-breakdown`, `budget`, `shrink`, `description`, `flush-latency`, `throughput`, `tradeoff`, `defrag-share`, `kappa`, `svg`, `svg-tradeoff`, `svg-pages` and `text-tradeoff`; a figure whose tables are missing is skipped. `svg-tradeoff` and `text-tradeoff` also draw the runs labelled `before/policy/knob/value/layout`, dashed and pale, to show how a change moved each policy's curve. `--summary` prints the last row of every series as a Markdown table. The figures are SVG, the same bytes whenever the data and matplotlib's version are, which the inline metadata of `tools/plot-evaluation.py` pins; `uv run` installs it.

Each evaluation page keeps its tables, gzipped, in `content/evaluation/data/`, and lists the commands that ran them and that redraw its figures.

27.8.2 The ripeness simulation

```
tools/simulate-ripeness.py compare|sweep|example [--set drafts|bayes] [options]
```

Simulates pages whose statements die at the rates of the shares they belong to, cleans each page when an estimator says it is ripe, and prints how much more each estimator costs than cleaning at the true ripeness, over all pages of a scenario. `compare` runs every estimator of a set, the ripeness drafts' or with `--set bayes` the Bayesian draft's; `sweep` varies their parameters `β`, `n0` and `c`; `example` computes the draft's worked example without noise. Its module docstring lists the estimators and the options.

27.8.3 Checks of this repository

command	checks
`tools/check-examples.py`	the Rust examples these pages mark with a `kladde-example` comment, compiled against a checkout of kladde-rs next to this one, and run, or refused by the compiler, where the comment says so
`tools/make-pdf.py --check-only`	the links and anchors between pages, and that `tools/pages.py` lists every page
`tools/make-pdf.py --self-test`	the layout passes of the diagrams
`site/build.sh`, then `tools/check-site-links.py`	the links and anchors of the built site, as Quartz made them

The README describes them in full.

27.8.4 Continuous integration

`.github/workflows/site.yml` runs on every push and pull request: it builds the PDF, builds the site and checks its links, and compiles and runs the examples against kladde-rs's `main`. A broken link fails the run; a failing example does not hold up publishing. On `main`, it then publishes the site and the PDF.

27.9 A measurement kept with its evaluation

The heap an open drawing takes was measured once, for the evaluation of variable-size values on this repository's branch `defrag-shifts`, by `content/evaluation/data/variable-size/memory.rs` there. It counts the bytes allocated through a global allocator while a drawing is parsed and while its kladde file is opened, and the median time of 21 opens, one drawing per call; it is built as the binary of a crate of its own that depends on kladde and kladde-svg, with their layout features.

Chapter 28

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 `τ = 0.8` is out of reach under the churn floor `λ = 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.

28.1 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 `Kladde<PersistableHashMap<u64, PersistableString>>` 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.

28.2 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.

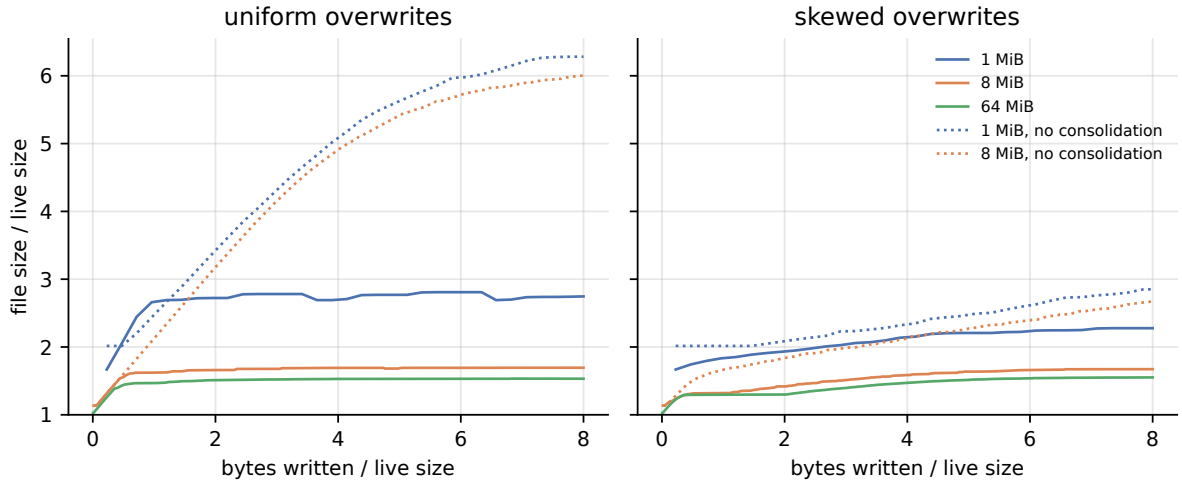


Figure 28.1: File size over live size as the overwrites accumulate, with and without consolidation.

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.

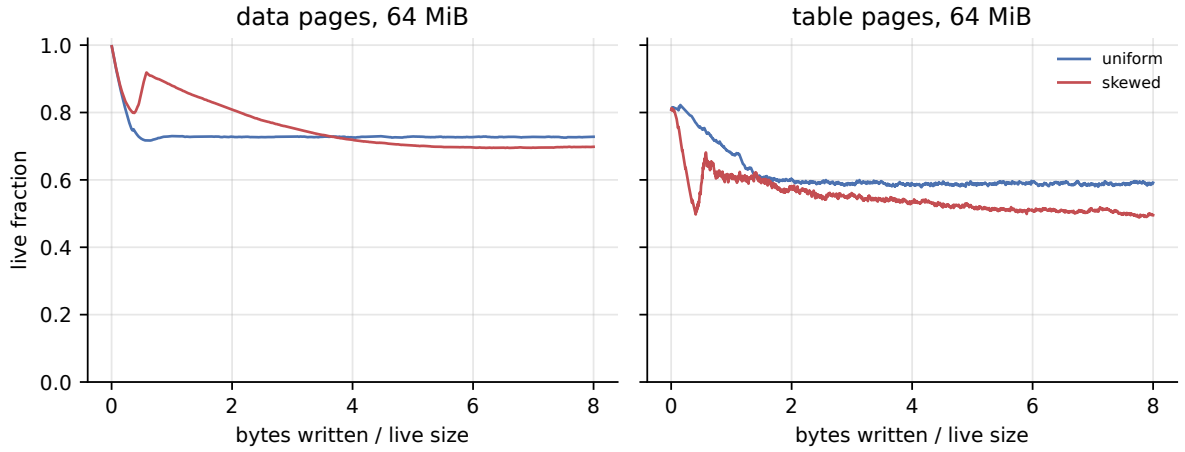


Figure 28.2: The live fraction of data pages and of address-table pages, in the 64 MiB files.

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 %.

28.3 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.

The fill the file reaches is then the churn floor's. A victim may be at most `1 / (1 + λ)` 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 + λ)` spends its time at an average fill of `λ / ((1 + λ) · ln(1 + λ))`, 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 `λ = 1`, the 8 MiB files reach a whole-file fill of 0.63, table pages and free pages included, well short of `τ = 0.8`.

The controller does reach targets that the churn floor allows, but by swinging between its bounds rather than settling. With `τ = 0.6`, the 8 MiB files end at whole-file fills of 0.58 (uniform) and 0.59

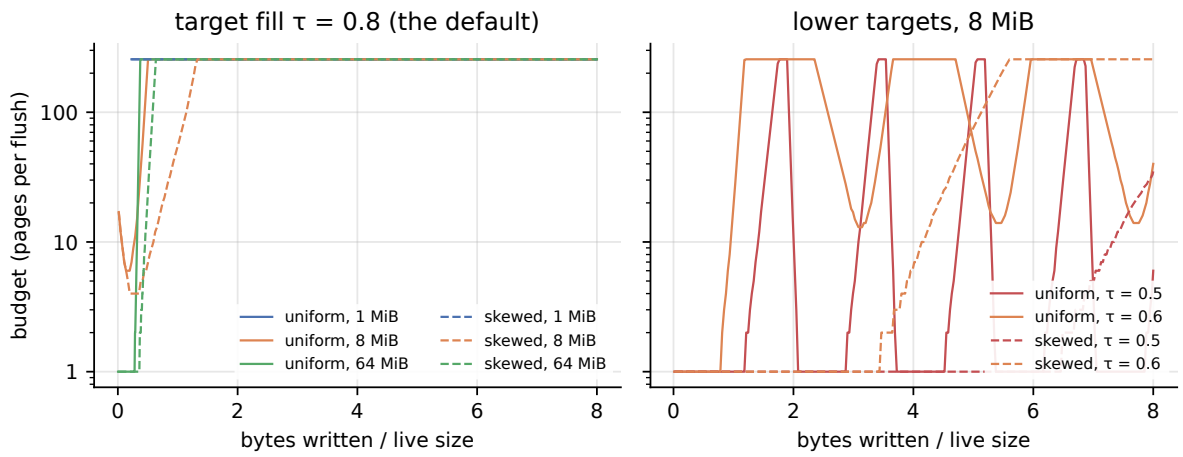


Figure 28.3: The budget, in pages per flush, with the default target fill (left) and with lower targets (right).

(skewed), and with `τ = 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 · (τ − 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.

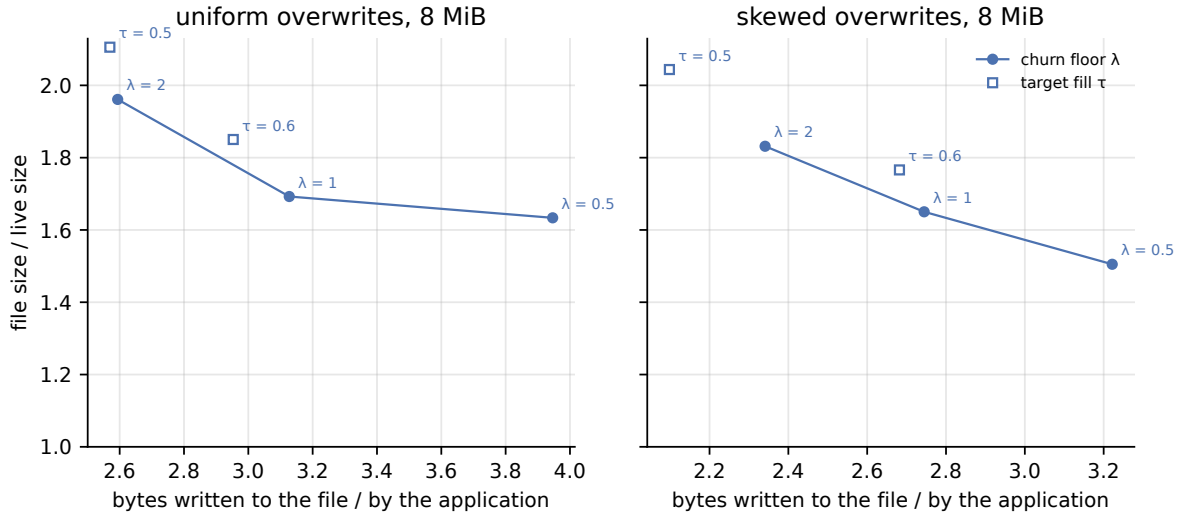


Figure 28.4: File size against bytes written in the steady state of the 8 MiB overwrite runs, the second half of each, for three churn floors and two lower targets.

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, `λ = 2` writes about as much as `τ = 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, `τ = 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.

28.4 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.

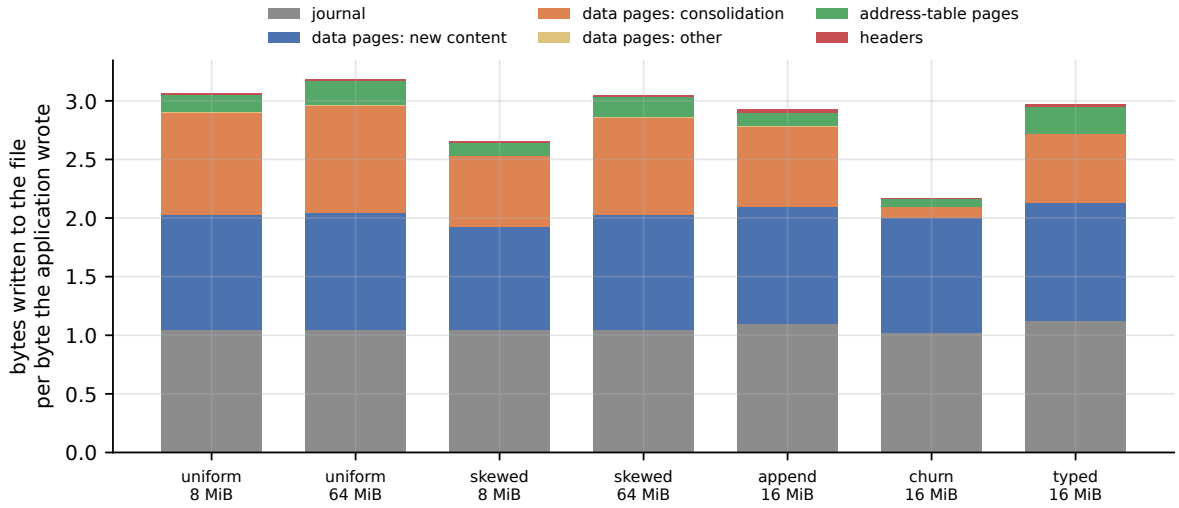


Figure 28.5: Where the bytes written to the file went, per byte the application wrote, over each whole run.

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 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.

28.5 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.

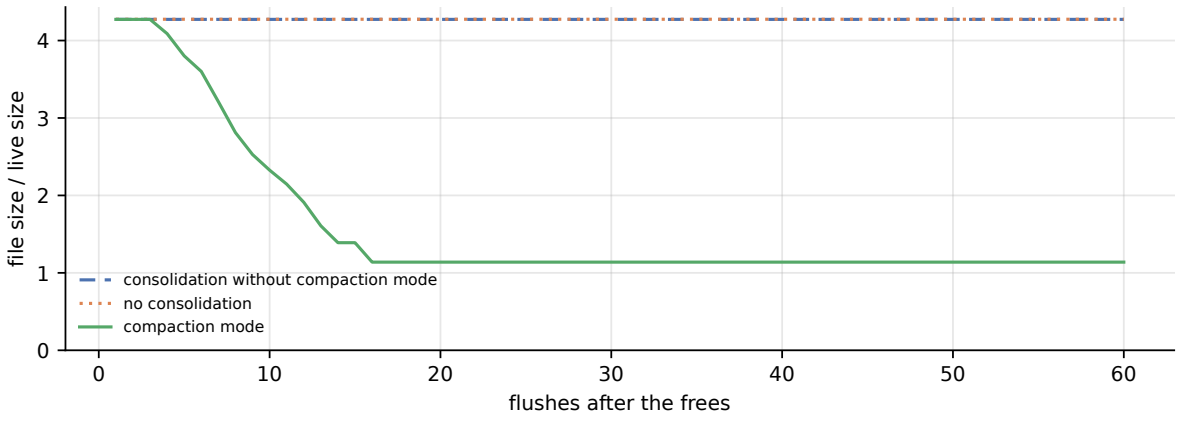


Figure 28.6: File size over live size after three quarters of the file were freed.

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.

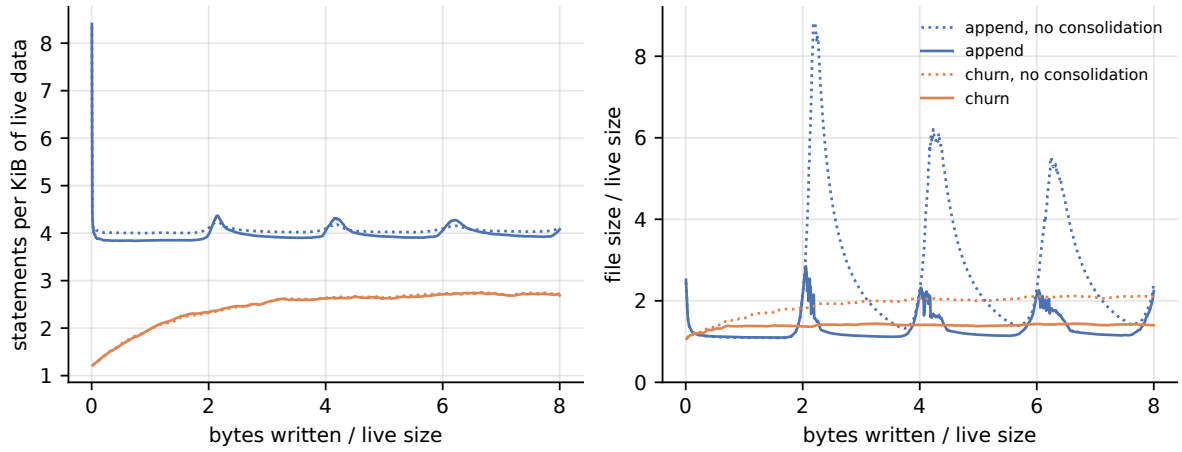


Figure 28.7: Statements per KiB of live data, and file size over live size, under appends and churn.

28.6 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.

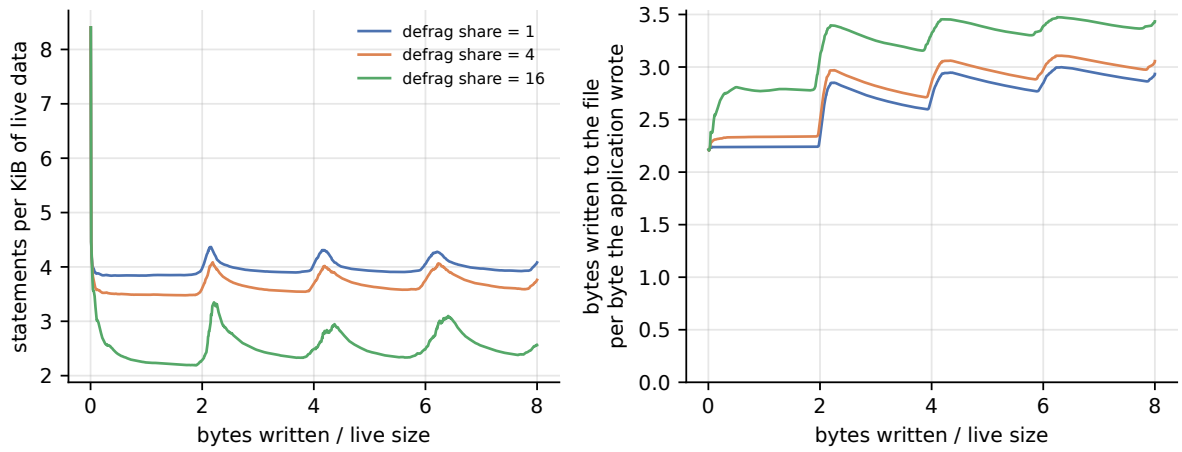


Figure 28.8: Statements per KiB of live data, and bytes written per byte, for three shares of description defragmentation.

28.7 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 µs each.

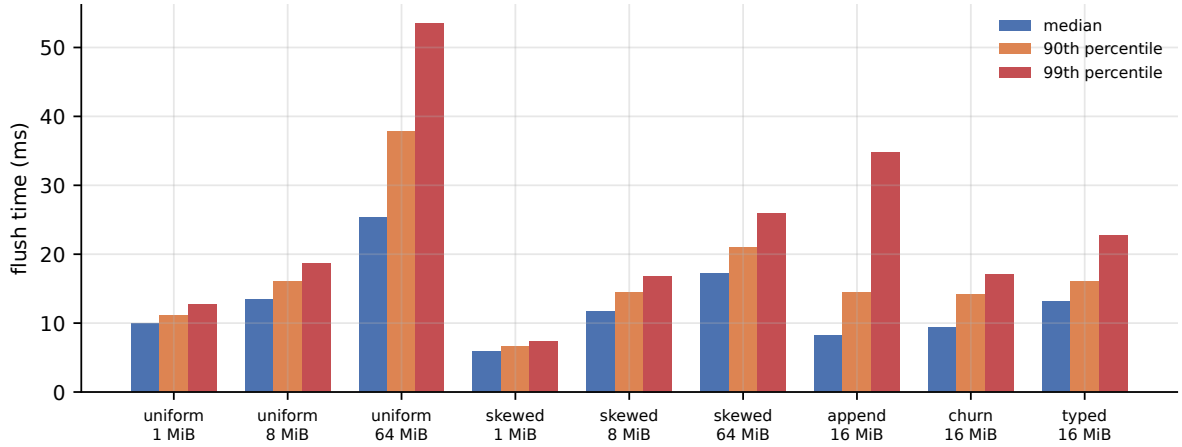


Figure 28.9: Flush times, including the fsync: median, 90th and 99th percentile.

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 µs, 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 28.10: 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.

28.8 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.

28.9 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 restatements 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.

28.10 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.

28.11 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 29

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.

29.1 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 `β = 0.1` per flush, a floor `R_MIN = 10⁻⁴` and a starting price `κ = 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 `τ = 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.

29.2 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.

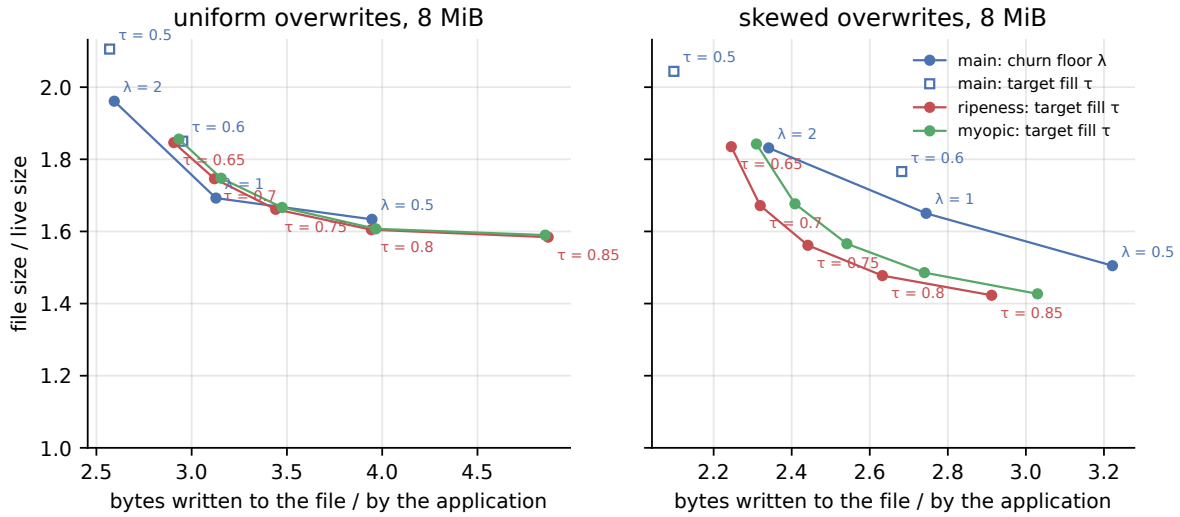


Figure 29.1: File size against bytes written in the steady state of the 8 MiB overwrite runs: main's churn floors and targets, ripeness's targets, and the same targets ranked without the logarithm ("myopic"), labelled where ripeness's are.

8 MiB, steady state	main	ripeness, same writes	ripeness, same file size
skewed, `λ = 2`	1.83 times the live size, 2.34 bytes per byte	1.65 (−10 %)	2.25 bytes per byte (−4 %)
skewed, `λ = 1`	1.65, 2.74	1.46 (−12 %)	2.34 (−15 %)
skewed, `λ = 0.5`	1.51, 3.22	below 1.42	2.57 (−20 %)
uniform, `λ = 1`	1.69, 3.13	1.74 (+3 %)	3.32 (+6 %)
uniform, `λ = 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

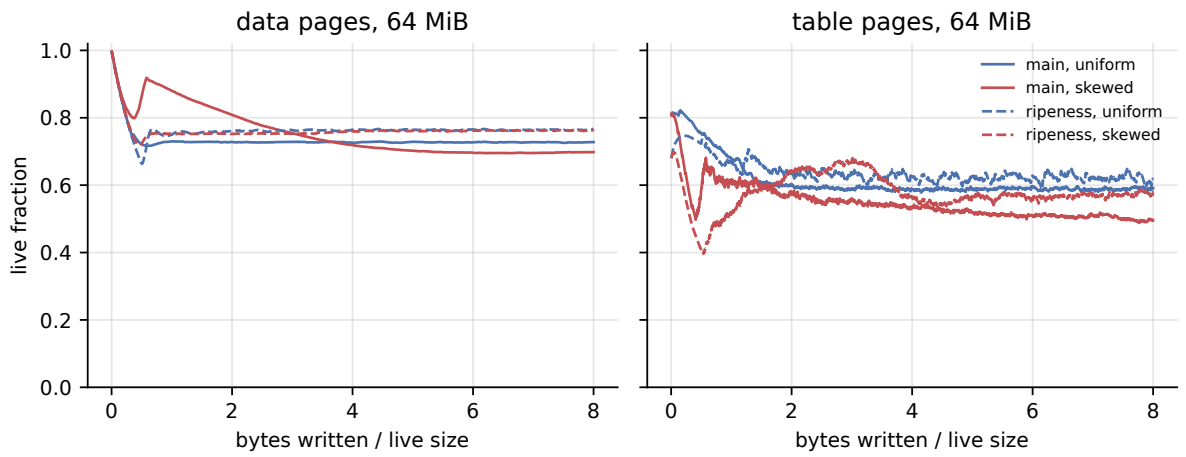


Figure 29.2: The live fraction of data pages and of address-table pages, in the 64 MiB files.

sporadic losses, so pages that drain alike get different estimates, and a threshold that differs page by page costs a little at light cleaning.

29.3 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(gain · (τ − 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.

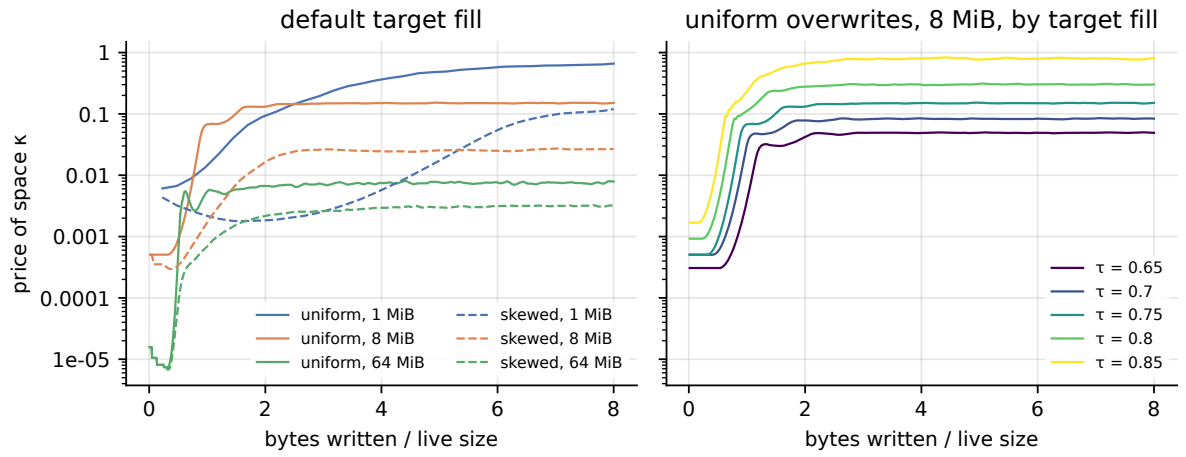


Figure 29.3: The price of space: the default runs (left), and the uniform 8 MiB runs by target (right).

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.

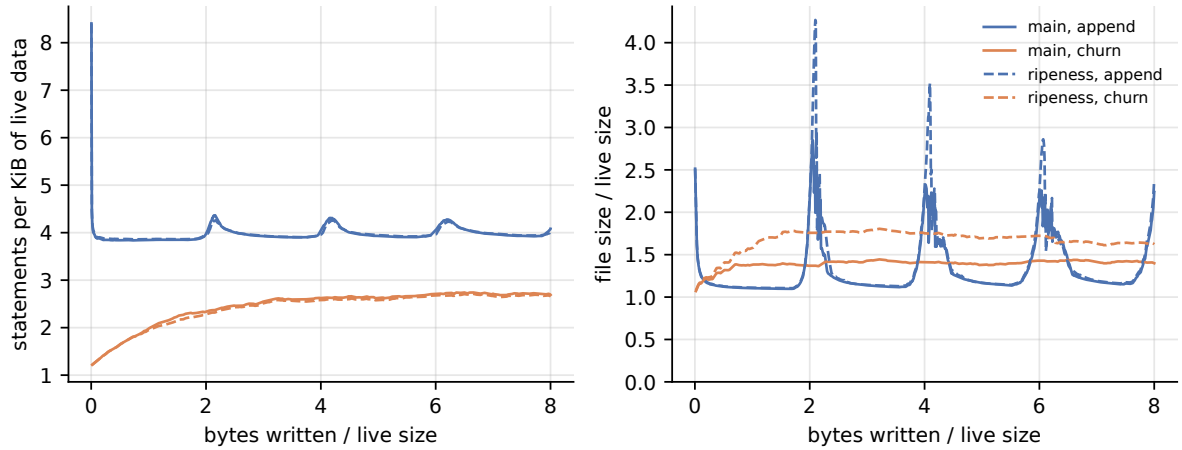


Figure 29.4: Statements per KiB of live data, and file size over live size, under appends and churn.

29.4 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.

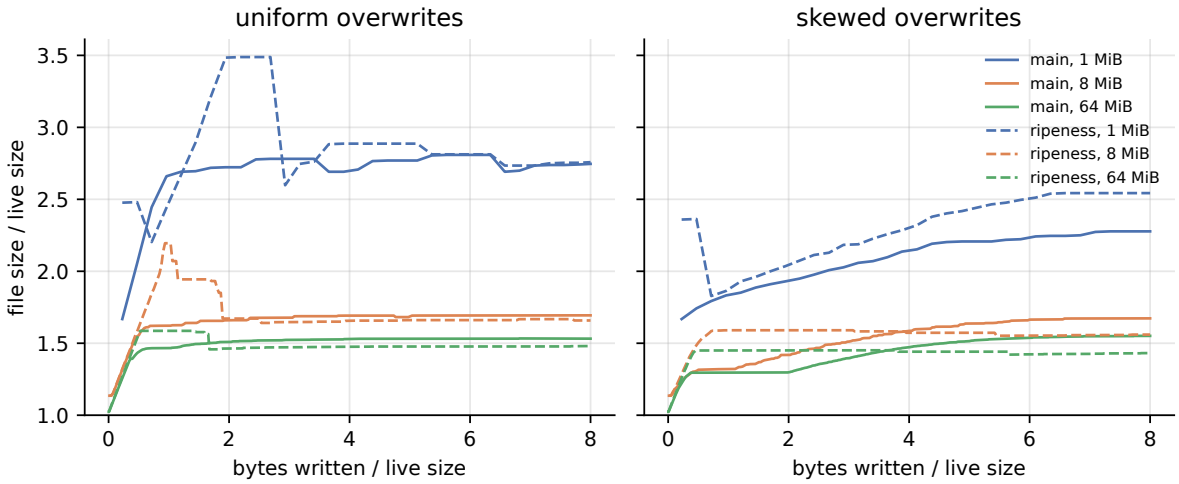


Figure 29.5: File size over live size as the overwrites accumulate.

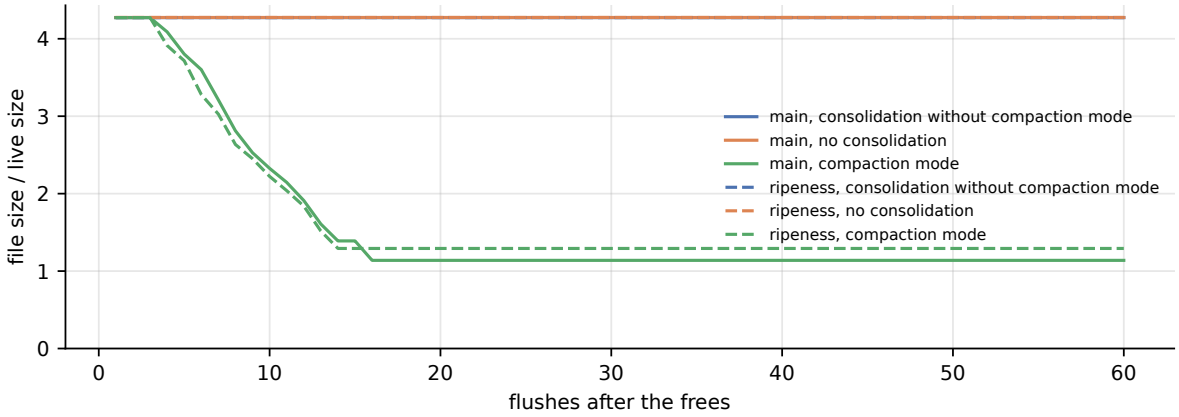


Figure 29.6: File size over live size after three quarters of the file were freed.

29.5 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.

29.6 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, `τ = 0.7`	1.67, 2.32	2.41 (+4 %)
skewed, `τ = 0.75`	1.56, 2.44	2.55 (+5 %)
skewed, `τ = 0.8`	1.48, 2.63	2.78 (+6 %)
uniform, `τ = 0.7`	1.75, 3.12	3.16 (+1 %)
uniform, `τ = 0.75`	1.66, 3.44	3.52 (+2 %)
uniform, `τ = 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 %

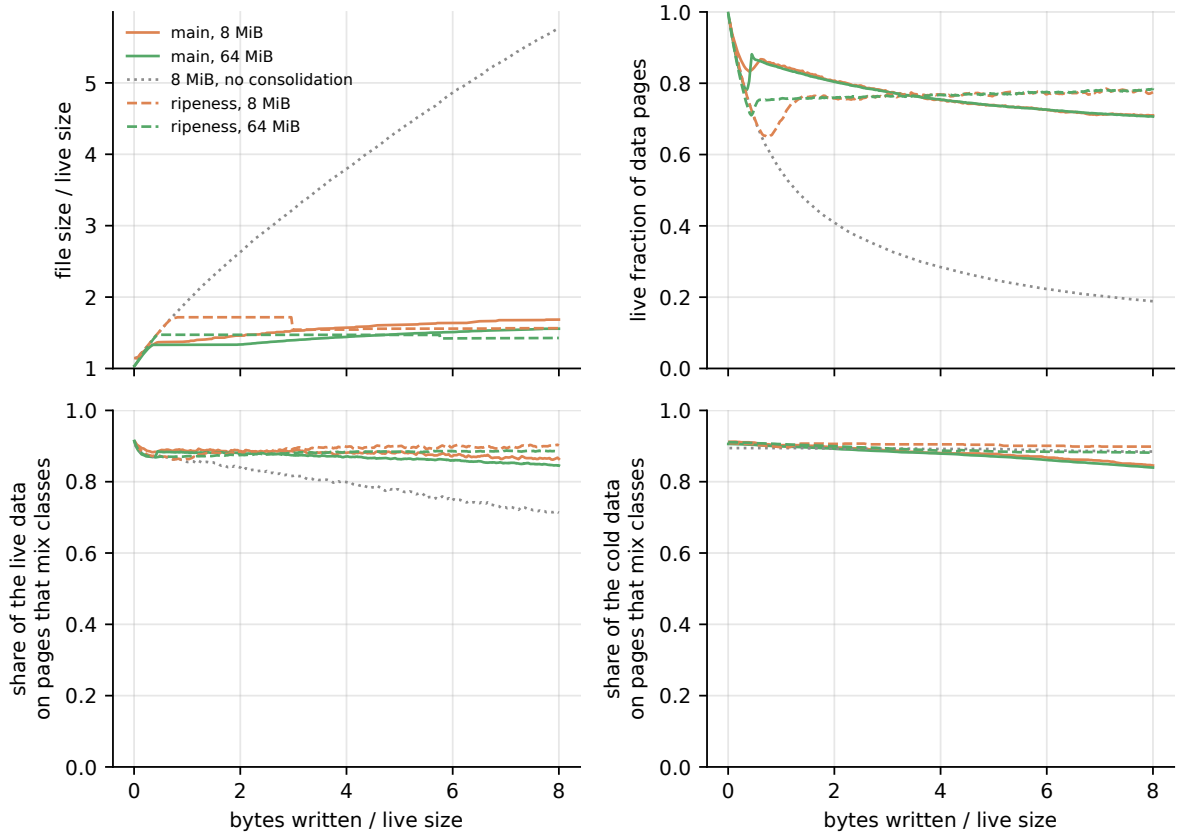


Figure 29.7: The mixed workload under main and ripeness: file size over live size, the live fraction of the data pages, the share of the live data on pages that mix classes, and the share of the cold data on such pages.

more, under churn 1.5 % more for a file 1 % smaller, and after the mass free it compacts exactly as the full rule does.

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.

29.7 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 `u₀` 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 29.8: 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.

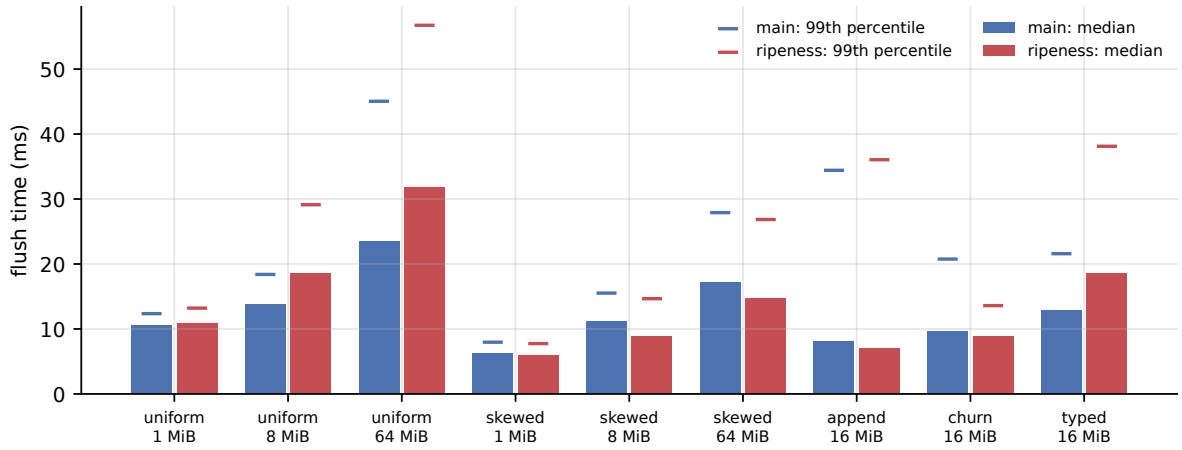


Figure 29.9: Flush times, including the fsync, in the first version's run: each run's median, and its 99th percentile.

29.8 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.

29.9 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 30

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.

30.1 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.

30.2 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.

30.3 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 %.

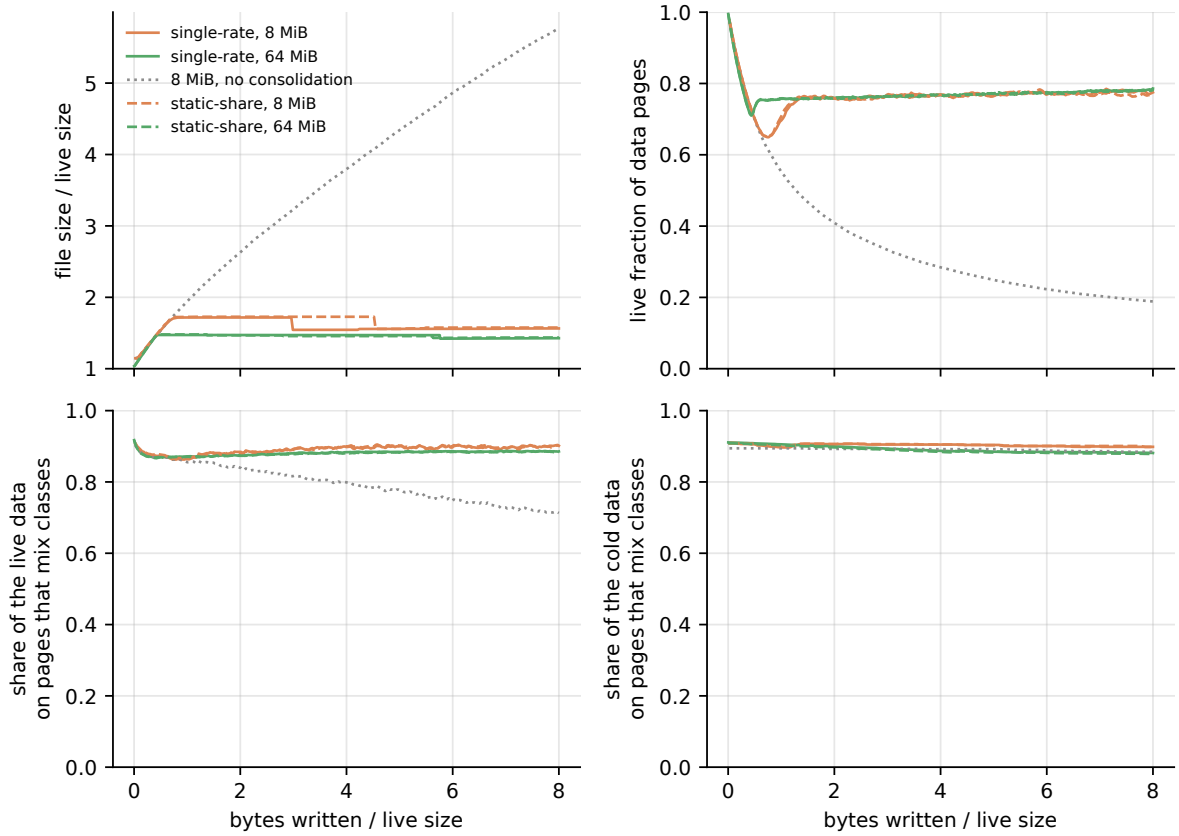


Figure 30.1: The mixed workload on both branches: file size over live size, the live fraction of the data pages, the share of the live data on pages that mix classes, and the share of the cold data on such pages.

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.

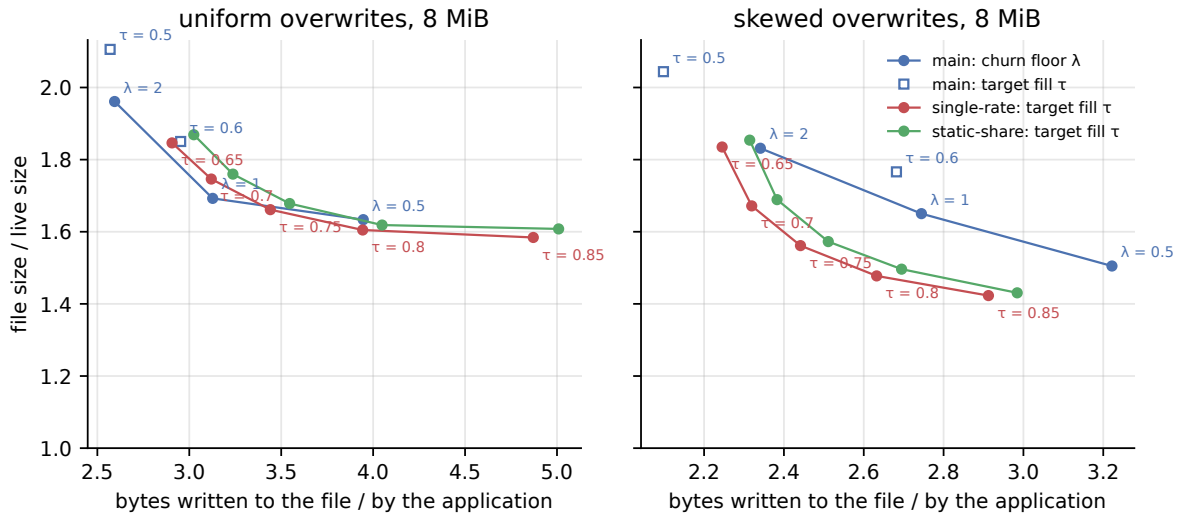


Figure 30.2: File size against bytes written in the steady state of the 8 MiB overwrite runs: main's churn floors and targets, and the two ripeness branches' targets, labelled where the single-rate branch's are.

30.4 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 `f32`s 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 %.

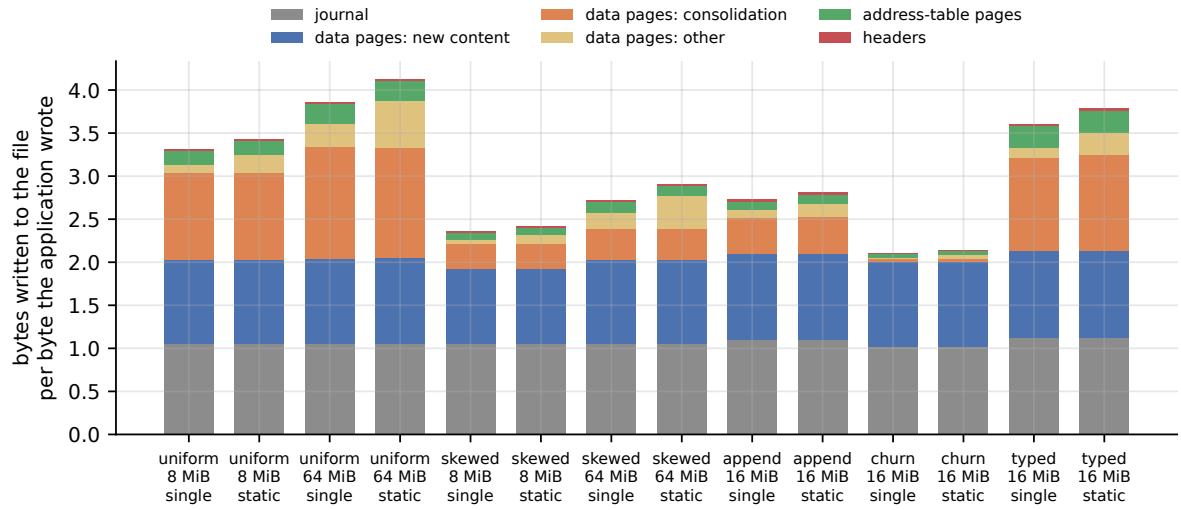


Figure 30.3: Where the bytes written to the file went, per byte the application wrote, over each whole run, on the single-rate branch and the static-share branch. "Other" is mostly the consolidator state.

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.

30.5 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.

30.6 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 31

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.

31.1 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, `β = 0.1` per flush, a class prior worth `ν_π = 10` chunks, and `R_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`.

31.2 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 %

with the state	single rate	static share	posterior, (a)	posterior, (c′)	(a) against single rate
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 %
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.

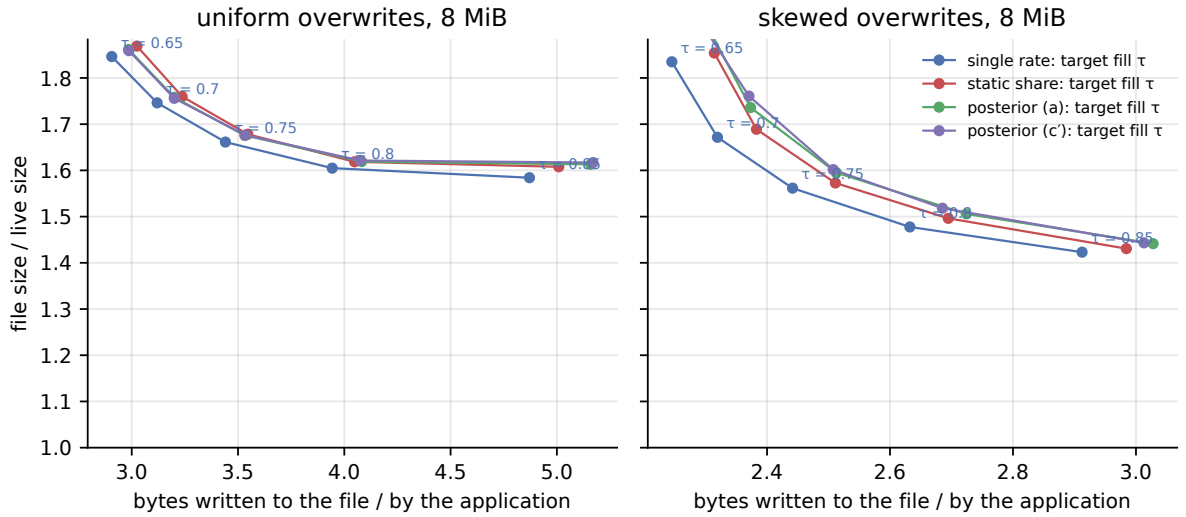


Figure 31.1: File size against bytes written in the steady state of the 8 MiB overwrite runs, with the state: the targets of the single-rate branch, the static-share branch, and the posterior under rules (a) and (c′), labelled where the single-rate branch's are.

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.

31.3 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 %
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 0.45^3. 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 %.

31.4 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

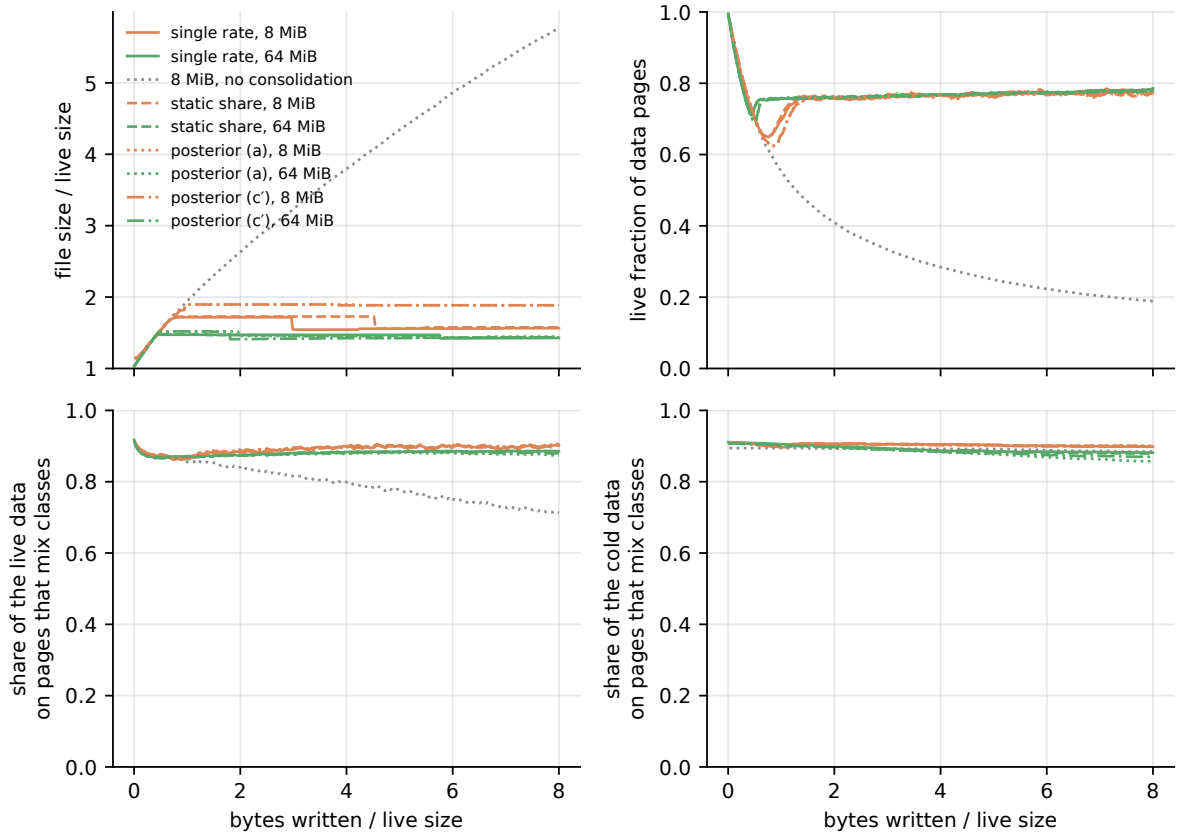


Figure 31.2: The mixed workload on the three branches, the posterior under both rules, with the state: file size over live size, the live fraction of the data pages, the share of the live data on pages that mix classes, and the share of the cold data on such pages.

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 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.

31.5 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 `f32`s, 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 %

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

flushes, uniform, 1 MiB	single rate	static share	posterior, first	posterior, faster ranking
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 `f32`s, 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.

31.6 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, $A_0 \le \Sigma\ k_i$, 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.

31.7 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
```

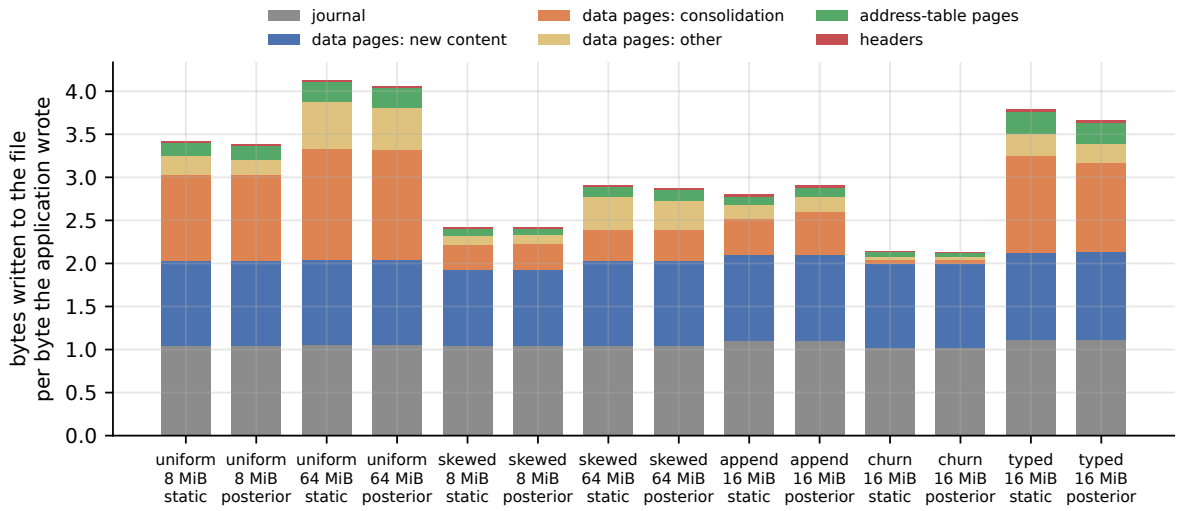


Figure 31.3: Where the bytes written to the file went, per byte the application wrote, over each whole run, on the static-share branch and the posterior under rule (a). "Other" is mostly the consolidator state.

The rerun at `08b575d` ran `./target/release/kladde-bench /tmp/bench-08b575d uniform skewed append 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 quic
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 32

Cleaning edited drawings and texts

32.1 Executive summary

Cleaning until its live pages are about nine tenths full, kladde keeps the live pages of an edited drawing at 1.12 to 1.35 times its live size and of a typed text at 1.10 to 1.19 times; at main's default settings they take 1.34 to 1.56 times the live size on the drawings, and 1.09 and 1.25 times on the texts. Four cleaning policies ran on kladde-svg's three drawings, in three layouts, under a drawing editor's edits that keep to one part of the drawing at a time, and on texts of 64 and 512 KiB, typed into keystroke by keystroke. In the regime this page dwells on, main's churn floor runs at $\lambda = 0.25$, which fills the drawings' live pages 89 to 92 % full. Cleaning by ripeness, with a static share, and by Bayesian ripeness each aim at a target fill of 0.9, which they reach on the world map and fall short of on the tiger and the coat of arms, at 80 to 90 %. The churn floor at $\lambda = 0.1$ fills the drawings' live pages fullest, 91 to 97 %, at 1.08 to 1.19 times the live size.

Main's churn floor lies lowest on every workload, or within 0.7 % of the lowest. At equal writes, the ripeness branches keep 0.3 to 15 % more live pages than the churn floor in the drawings' base and packed layouts, between 0.7 % fewer and 8 % more in the small layout, and 2 to 22 % more on the texts. The churn floor also reaches further: at $\lambda = 0.1$ it keeps 1.08 to 1.19 times the drawings' live size, where the best setting of any ripeness branch stops at 1.10 to 1.26. In the regime, the ripeness branches keep 1 to 12 % more live pages than the churn floor at $\lambda = 0.25$, with between 19 % fewer and 10 % more writes.

The ripeness branches give up the cleaning that costs nothing. Main's free filling takes any victim that fits into the room of the pages a flush writes anyway: at $\lambda = 1$, 54 to 919 per run, half to four times as many as the pages it opens to clean, 14 to 1,673. The ripeness branches measured here take only small, ripe victims there, 0 to 116 per run at $\tau = 0.9$, and open 117 to 4,809 pages instead. With main's free filling instead, the ripeness branches close a third of their gap to the churn floor in the drawings' base and packed layouts, two thirds in the small layout, and none of it on the texts.

The leaves fill about as much as the data pages under the churn floor, and a few points less under the ripeness branches. In the drawings' base and packed layouts, the address table's statements are 19 to 27 % of the live bytes, and the leaves that hold them 20 to 29 % of the live pages. At $\lambda = 0.25$, the leaves come out 88 to 90 % full, against data pages 89 to 93 % full, since the churn floor prices a leaf as it prices a data page; under the ripeness branches at $\tau = 0.9$, they come out 79 to 86 % full, against data pages 82 to 93 %. Leaves as full as the data pages would save at most 3.5 % of the live pages. In the drawings' small layout and on the texts, the leaves are 3 to 7 % of the live pages, and their fill matters little.

Free pages make up much of the smaller files, and cleaning harder adds to them. During the edits, the tiger's files hold 23 to 47 free pages beside 13 to 34 live ones, and the coat of arms' 26 to 71 beside 35 to 95; the texts' free pages are 20 to 30 % of the file at 64 KiB and 11 to 12 % at 512 KiB. Closed, the tiger's files take 2.4 to 5.1 times its edited SVG, the coat of arms' 1.5 to 3.0 times, the world map's 1.07 to 1.9 times, and the texts' 1.24 to 1.71 times the typed text. From $\lambda = 1$ to $\lambda = 0.1$, the drawings' files gain 3 to 41 free pages while they lose 3 to 99 live ones, and the closed files come out between 16 % smaller and 24 % larger.

Filling the pages costs 1.2 to 1.9 times the writes of main's default on the drawings, and up to 1.2 times

on the texts, which is still a fraction of saving the file. In the regime, an edit writes 118 to 432 bytes, against 96 to 233 at main's default, and a keystroke 40.5 to 54.2, against 36.8 and 49.9. Saving the canonical SVG every 100 edits would write 590 bytes per edit on the tiger, 1,845 on the coat of arms, and 10,699 on the world map, and saving the text every 1000 keystrokes 241 and 700 bytes per keystroke. On the drawings, the journal and the headers take 48 to 83 bytes per edit under every policy, and cleaning decides the rest; on the texts, the journal, the headers, and the leaves that splices fill with restated statements take 31 to 47 bytes of each keystroke's writes, and the data pages 5 to 13.

Edits and keystrokes stay fast, and flushes short: in the regime, kladde's calls take 22,000 to 108,000 edits per second on the drawings, against 37,000 to 119,000 at main's default, and 936,000 to 1,016,000 keystrokes per second on the texts; the median flush takes 3.2 to 10.6 ms on the drawings and 6.8 to 10.8 ms on the texts. On the drawings, the regime costs up to 42 % of the edit rate, the least under single-rate ripeness, up to 17 %; on the world map, the static share's and the Bayesian posterior's flushes take 1.2 to 1.7 times as long as single-rate ripeness's. Reopening an edited file takes 0.7 to 18 ms.

32.2 What was measured

kladde-svg's `svg-bench` edits an SVG drawing stored in kladde as a drawing editor would, recording a row in kladde-bench's format every 1000 edits. Its mix: 30 % drags, which set a shape's translation; 20 % node moves, 8 % node insertions and 7 % node deletions in a path's data; 15 % recolours; 8 % duplications and 7 % deletions of a shape; and 5 % restacks, which move a shape among its siblings. Nine edits in ten go to a window of a tenth of what they pick from, the shapes, the paths, or the elements without children, at the same place in each, and the window jumps elsewhere every 3000 edits, so that the edits keep to one part of the drawing at a time, as a person's do. This is `svg-bench`'s default since kladde-rs `e88fd47`.

The drawings are the Ghostscript tiger, the coat of arms of Spain, and a blank world map, each edited as many times as it took an earlier, uniform mix to store eight times the drawing's live size in the base layout:

drawing	edits	canonical SVG, before → after the edits	live size during the edits: base / packed / small
tiger	31,613	66,850 → 59,005 bytes	92 / 47 / 46 KiB
coat of arms	41,130	163,768 → 184,541 bytes	252 / 131 / 128 KiB
world map	177,831	1,005,908 → 1,069,891 bytes	1,353 / 954 / 943 KiB

Every run makes the same edits, whatever the branch, layout, or setting. The *live size* is the bytes of the drawing's allocations, and the *canonical SVG* what kladde-svg writes, without whitespace between tags and with path data in the grammar's shortcuts; the edits leave the coat of arms and the world map larger, and the tiger smaller.

The layouts are cargo features of kladde-svg that choose the model's containers: *base* stores every list as a vector of fixed slots, *packed* packs the lists of enums, children, transforms, attributes and path data, and *small* also keeps short lists and strings inline in their parent. The small layout keeps most of a drawing's content in a few large allocations, so that its address table is small.

kladde-bench's `text-bench` types into one text, stored in a single allocation of the store, keystroke by keystroke at a cursor, as a text editor would, recording a row every 1000 keystrokes; Benchmarks and tests describes it. Of its 100,000 keystrokes, 85 % type a letter, 10 % delete the one before the cursor, 4.5 % move the cursor, nine moves in ten within a window of a tenth of the text that jumps every 5000 keystrokes, and 0.5 % paste 200 letters; every keystroke that changes the text is a splice, which the flush states by restating the statements behind it. It ran on texts of 64 and 512 KiB of random words, which the typing grows to 241,322 and 700,340 bytes; their live size averaged 193 and 642 KiB over the second half.

The store flushes whenever the journal outgrows four pages, `Options::journal_budget_pages = 4`, and at every row: on the drawings every 230 to 372 edits on average, on the texts every 1000 keystrokes, the journal never filling between rows; otherwise it runs with its defaults.

Four branches of kladde-rs ran, each with the newest size rule:

branch	commit	policy	knob, default	settings run
`defrag-shifts`	`7eb2ec4`	main's churn floor	churn floor λ, 1	1, 0.5, 0.35, 0.25, 0.15, 0.1
`ripeness-svg`	`d3b3a7e`	cleaning by ripeness, one rate per page	target fill τ, 0.75	0.75, 0.85, 0.9, 0.95
`ripeness2-svg`	`f659de7`	ripeness with a static share per page	target fill τ, 0.75	as above
`bayesian-ripeness-svg`	`65d041a`	Bayesian ripeness, rule (a), with the empirical prior capped at the events it has seen	target fill τ, 0.75	as above

The three ripeness branches are ports, onto `defrag-shifts`, of the branches that Cleaning by ripeness, Ripeness with a static share, and Bayesian ripeness measured on kladde-bench's workloads. On every branch, a budgeted rewrite of leaves opens no page of its own and costs the budget the pages its restatements take, which the page rewrite estimates at the leaves' coverage; on main, the churn floor weighs a leaf by the same estimate, as the budget loop has it. `text-bench` came from kladde-rs `7c5cdd6`, on the branch `text-bench`, and was added to each of these commits with `git cherry-pick`, without its `--churn-floor` option on the ripeness branches, whose store lacks it.

Main's budget controller aims the file at a target fill of 0.8, measured over all of its pages, free ones included. Over the second half of every run but those on the 512 KiB text, it held the budget at its maximum, 256 pages, but for a row or two in four runs on the world map, where it dipped to 196 pages at the lowest, so that the churn floor alone decided what was cleaned; on the world map, whose file starts out fuller than the target, it held the budget lower during parts of the first half. On the 512 KiB text, the file passed the target after about 11,000 keystrokes, the controller had cut the budget to its minimum, one page, by 39,000 to 42,000, and every λ cleaned about alike. The ripeness branches' controller moves the price of space until the data pages and leaves are as full as the target.

The regime this page dwells on is live pages about nine tenths full: $\lambda = 0.25$ for the churn floor, and $\tau = 0.9$ for the ripeness branches, whose results at 0.85, 0.9, and 0.95 hardly differ on the tiger and the coat of arms. Main's default, $\lambda = 1$, is the reference, and $\lambda = 0.1$, which fills the pages most, the churn floor's furthest reach; the sweep has every setting. The tables name the churn floor by its λ, and the three ripeness branches, each at $\tau = 0.9$, by their estimate: *ripeness*, *static share*, and *Bayesian*.

Space is averaged over the second half of each run, and writes are what kladde wrote during that half, per edit or keystroke, as on the other evaluation pages. *Live pages* are the data pages and the leaves, the latter with the interior pages of the address table, but without the header and the journal. *Fill* is the live bytes in a page over the 4,081 bytes it can hold: for a data page the bytes that fragments resolve to, for a leaf its statements' framing and inline payloads. *Free pages* are pages within the file that hold nothing live; the *closed file* is the file's size after the last edit and `close`, which truncates it after its last live page. Writes count the journal's bytes and every page written, headers included, at 4,096 bytes each.

The space and the writes come from 198 runs, eight at a time, 162 of `svg-bench` and 36 of `text-bench`; the times from two further passes, one run at a time on an otherwise idle host, of the six configurations in the tables on the drawings and of every setting on the texts. All were built with `--release` by Rust 1.99.0 and ran on an Intel Core i7-1165G7, four cores and eight threads, with 16 GB of memory under Linux 7.0, in a development container, on mains power. The benchmarks are deterministic but for their times, and both timing passes reproduced the first runs' tables to the byte; the times give their mean, and how far the two passes differed. `data/cleaning-edits/` keeps every run's table, `logs/` beside them the end of each run's output, with the closed file's size, and `timing/` and `timing2/` that of the two timing passes, with the texts' tables, from which their keystroke rates come.

To reproduce a run, build `svg-bench` on the branch's commit, with `--features packed` or `--features small` for those layouts, and run it on the drawing with its number of edits, or add `text-bench` and run it on a text of the size:

```
git cherry-pick --no-commit 7c5cdd6   # text-bench; drop --churn-floor on the ripeness branches
cargo build --release -p kladde-svg --bin svg-bench --features packed
```

```
# --target-fill 0.9 instead of --churn-floor 0.25 on the ripeness branches
target/release/svg-bench crates/kladde-svg/corpus/drawings/world-map.svg \
    out/ --edits 177831 --journal-budget 4 --churn-floor 0.25
cargo build --release -p kladde-bench --bin text-bench
target/release/text-bench out/ --size 524288 --journal-budget 4 --churn-floor 0.25
```

The figures come from the tables in `data/`; each run's directory is named `<policy>-<knob><value>-<layout>`, with `text` for the layout of the texts' runs, and `tools/plot-evaluation.py` takes it labelled `<policy>/<knob>/<value>/<layout>`:

```
D=content/evaluation/data/cleaning-edits; F=content/evaluation/figures/cleaning-edits
runs=()
for dir in $D/*-*-*/; do
    IFS=- read -r p k l <<< "$(basename $dir)"; knob=${k%%[0-9]*}
    runs+=("$p/$knob/${k#$knob}/$l=$dir")
done
tools/plot-evaluation.py --only svg-tradeoff,text-tradeoff --out $F "${runs[@]}"
tools/plot-evaluation.py --only svg-pages --out $F $(printf '%s\n' "${runs[@]}" |
    grep -E '^(main/lambda/(1|0.25|0.1)|[a-z0-9]+/tau/0.9)/')
```

32.3 Live pages

On the drawings, the regime takes 9 to 21 % of main's live pages at its default away, most of them data pages; on the 64 KiB text, 5 to 6 %, while on the 512 KiB text, which main's default already keeps 95 % full, the churn floor at $\lambda = 0.25$ keeps as many and the ripeness branches 7 to 8 % more. Each cell gives the live pages, and in parentheses the leaves among them; the first column gives the live bytes they hold, in data pages and in leaves, which differ by at most 1.4 % between the policies:

workload	layout or size	live bytes, KiB	$\lambda = 1$	$\lambda = 0.25$	$\lambda = 0.1$	ripeness	static share	Bayesian
tiger	base	72 + 26	34.2 (8.8)	26.9 (7.1)	25.8 (6.9)	30.1 (8.2)	28.4 (8.2)	29.3 (8.0)
tiger	packed	37 + 14	18.0 (4.6)	14.4 (4.0)	14.1 (4.0)	15.8 (4.4)	14.9 (4.4)	15.9 (4.4)
tiger	small	45 + 1	16.1 (0.5)	13.1 (0.5)	12.7 (0.5)	14.3 (0.5)	13.2 (0.5)	14.6 (0.6)
coat of arms	base	203 + 67	94.8 (23.7)	75.2 (18.8)	71.0 (17.7)	78.2 (20.3)	76.6 (20.5)	77.1 (20.0)
coat of arms	packed	107 + 38	51.0 (13.6)	40.6 (10.8)	38.3 (10.1)	43.2 (11.9)	40.9 (11.5)	42.0 (11.0)
coat of arms	small	127 + 7	45.8 (2.1)	37.1 (2.0)	35.0 (2.0)	40.3 (2.6)	37.5 (2.8)	40.8 (2.8)
world map	base	1,110 + 345	478.8 (123.6)	398.6 (96.6)	379.4 (91.6)	408.0 (106.3)	409.8 (106.1)	409.2 (107.1)
world map	packed	834 + 202	333.2 (72.6)	282.2 (57.0)	269.3 (53.8)	290.0 (63.3)	291.3 (63.5)	290.8 (62.6)
world map	small	937 + 36	314.1 (12.0)	264.6 (10.6)	253.8 (11.1)	273.1 (15.1)	274.1 (17.1)	273.7 (14.3)
text	64 KiB	192 + 13	60.4 (3.8)	56.9 (3.8)	54.3 (3.7)	57.4 (3.9)	57.5 (4.1)	57.3 (3.8)
text	512 KiB	640 + 28	175.7 (7.6)	176.5 (7.6)	175.2 (7.6)	188.5 (10.7)	187.9 (11.3)	189.9 (10.6)

Over the live size, which the first column gives here, the same live pages:

workload	layout or size	live size, KiB	$\lambda = 1$	$\lambda = 0.25$	$\lambda = 0.1$	ripeness	static share	Bayesian
tiger	base	92	1.49	1.17	1.12	1.31	1.24	1.28
tiger	packed	47	1.52	1.22	1.19	1.33	1.26	1.35
tiger	small	46	1.39	1.12	1.09	1.23	1.14	1.26
coat of arms	base	252	1.50	1.19	1.13	1.24	1.22	1.22
coat of arms	packed	131	1.56	1.24	1.17	1.32	1.25	1.29
coat of arms	small	128	1.43	1.16	1.09	1.26	1.17	1.27
world map	base	1,353	1.42	1.18	1.12	1.21	1.21	1.21
world map	packed	954	1.40	1.18	1.13	1.22	1.22	1.22
world map	small	943	1.34	1.12	1.08	1.16	1.16	1.16
text	64 KiB	193	1.25	1.18	1.13	1.19	1.19	1.19
text	512 KiB	642	1.09	1.10	1.09	1.17	1.17	1.18

The live bytes exceed the live size in the drawings' base and packed layouts, since the statements that describe the allocations are live too.

The churn floor fills the drawings' data pages 89 to 93 % full at $\lambda = 0.25$ and 93 to 97 % at $\lambda = 0.1$; the ripeness branches at $\tau = 0.9$ fill them 82 to 93 % full. Each cell gives the fill of the data pages, of the leaves, and of all live pages, in percent:

workload	layout or size	$\lambda = 1$	$\lambda = 0.25$	$\lambda = 0.1$	ripeness	static share	Bayesian
tiger	base	71 / 74 / 71	91 / 90 / 91	96 / 93 / 95	83 / 79 / 82	90 / 79 / 87	85 / 81 / 84
tiger	packed	70 / 75 / 71	89 / 88 / 89	93 / 86 / 91	82 / 80 / 81	90 / 79 / 87	82 / 80 / 81
tiger	small	73 / 28 / 72	91 / 33 / 89	94 / 35 / 91	83 / 28 / 81	90 / 29 / 88	82 / 32 / 80
coat of arms	base	72 / 71 / 72	90 / 89 / 90	96 / 94 / 95	89 / 83 / 87	92 / 82 / 89	90 / 84 / 88
coat of arms	packed	72 / 71 / 72	90 / 89 / 90	96 / 93 / 95	86 / 81 / 85	92 / 83 / 90	88 / 86 / 87
coat of arms	small	73 / 80 / 73	91 / 81 / 90	97 / 85 / 96	85 / 64 / 84	92 / 65 / 90	84 / 63 / 83
world map	base	78 / 70 / 76	92 / 89 / 92	97 / 95 / 96	93 / 83 / 90	92 / 83 / 90	93 / 82 / 90
world map	packed	80 / 70 / 78	93 / 89 / 92	97 / 95 / 97	93 / 81 / 90	93 / 80 / 90	92 / 81 / 90
world map	small	78 / 76 / 78	93 / 91 / 92	97 / 94 / 97	91 / 66 / 90	92 / 59 / 90	91 / 70 / 90
text	64 KiB	85 / 87 / 85	91 / 87 / 90	95 / 88 / 95	90 / 81 / 90	91 / 74 / 89	90 / 84 / 90

workload	layout or size	λ = 1	λ = 0.25	λ = 0.1	ripeness	static share	Bayesian
text	512 KiB	96 / 93 / 95	95 / 93 / 95	96 / 94 / 96	90 / 65 / 89	91 / 61 / 89	90 / 65 / 88

On the world map, the ripeness branches reach their target, their live pages 90 % full at τ = 0.9 in every layout; on the tiger and the coat of arms, single-rate ripeness and the Bayesian posterior stop at 80 to 88 %, and the static share at 87 to 90 %. Their controller raises the price of space as long as the pages are emptier than the target, but on the smaller files no price makes it clean the pages that the churn floor cleans, and the targets 0.85, 0.9, and 0.95 end within a few points of each other. Of the three, single-rate ripeness writes least, and on the world map it fills the pages as full as the other two.

What the ripeness branches do not take is free cleaning. Free filling puts victims into the room left in the pages a flush writes anyway, which costs no page of its own. Main takes any whole victim that fits there, sparsest first; the ripeness branches measured here take only ripe victims whose survivors fill at most θ, 5 %, of a page, and then the survivors of a cursor through the previous flush's least-filled page, so that survivors, cold by selection, do not settle next to fresh content, as the ripeness draft proposed before it took main's free filling. So the ripeness branches clean nearly everything on pages they open for it. Each cell gives, over a whole run, the data victims that free filling took, the data pages opened only to take victims, and the leaves rewritten, by the budget or by the cut's fillers:

workload	layout or size	λ = 1	λ = 0.25	λ = 0.1	ripeness	static share	Bayesian
tiger	base	112 / 145 / 219	66 / 545 / 411	78 / 923 / 601	7 / 340 / 292	5 / 524 / 310	6 / 395 / 324
tiger	packed	90 / 64 / 118	56 / 301 / 200	51 / 482 / 275	8 / 196 / 168	1 / 336 / 168	0 / 213 / 175
tiger	small	97 / 120 / 61	66 / 413 / 65	59 / 664 / 63	8 / 241 / 61	5 / 415 / 61	7 / 245 / 64
coat of arms	base	215 / 226 / 388	131 / 1,045 / 858	100 / 2,282 / 1,569	9 / 847 / 645	11 / 1,155 / 660	15 / 1,004 / 737
coat of arms	packed	123 / 145 / 223	90 / 604 / 460	84 / 1,186 / 748	13 / 433 / 348	13 / 701 / 390	20 / 495 / 443
coat of arms	small	147 / 225 / 202	98 / 883 / 249	77 / 1,589 / 255	11 / 536 / 219	10 / 931 / 229	6 / 552 / 208
world map	base	919 / 1,194 / 3,002	673 / 4,729 / 6,522	646 / 10,636 / 12,451	72 / 4,405 / 5,112	88 / 4,630 / 5,657	96 / 4,720 / 5,256
world map	packed	795 / 817 / 1,792	573 / 3,130 / 3,867	523 / 7,691 / 7,993	50 / 3,102 / 2,865	97 / 3,269 / 3,050	60 / 3,173 / 3,018
world map	small	774 / 1,673 / 2,400	571 / 5,996 / 4,157	515 / 12,072 / 5,903	76 / 4,406 / 2,435	116 / 4,809 / 2,380	87 / 4,357 / 2,870
text	64 KiB	54 / 14 / 157	50 / 69 / 161	45 / 210 / 162	8 / 117 / 127	8 / 148 / 119	4 / 133 / 134
text	512 KiB	63 / 90 / 174	57 / 101 / 190	61 / 110 / 186	6 / 137 / 22	9 / 178 / 8	6 / 146 / 29

At λ = 1, main takes half to four times as many victims free as it opens pages for; the ripeness branches at τ = 0.9 take 0 to 16 % as many free as main does at λ = 1. That count leaves out the pages the cursor empties, a whole statement at a time, in the room that small ripe victims leave. Ripeness with main's

free filling gives the ripeness branches main's free filling, which takes about one victim per flush on the drawings, as main's does: they write 2 to 3 % less at the same target, and close a third to two thirds of their gap to the churn floor on the drawings, and none of it on the texts.

32.4 Leaves

Under the churn floor, the leaves of the drawings' base and packed layouts fill about as much as the data pages: 70 to 75 % at $\lambda = 1$, 88 to 90 % at $\lambda = 0.25$ and 86 to 95 % at $\lambda = 0.1$, where the data pages are 70 to 80 %, 89 to 93 % and 93 to 97 % full. The churn floor prices a leaf as it prices a data page: rewriting it reclaims the page less what the leaf's statements take restated, and writes those, which the page rewrite estimates at the leaf's coverage, so that a leaf is a victim up to the same fill as a data page. In those layouts, the statements are 19 to 27 % of the live bytes and the leaves 20 to 29 % of the live pages, so their fill matters; leaves as full as the data pages would save at most 2.8 % of the live pages under the churn floor.

Under the ripeness branches, the leaves stay a few points emptier than the data pages: at $\tau = 0.9$, 79 to 86 % full, against 82 to 93 %, and leaves as full as the data pages would save at most 3.5 % of the live pages. They charge a rewrite of leaves to the budget by the same estimate as main, but rank leaves by their ripeness; this page has not traced what keeps them emptier.

Where the leaves hold little, their fill matters little. In the drawings' small layout, the statements are 1 to 5 % of the live bytes and the leaves 3 to 7 % of the live pages, and on the texts 4 to 6 % and 4 to 7 %. On the 512 KiB text, the ripeness branches' leaves stay 61 to 65 % full; they rewrite 8 to 29 leaves over a whole run, and this page has not traced why so few.

32.5 The trade-off between space and writes

Each policy trades writes for space along its own curve, and the churn floor's lies lowest on every workload, or within 0.7 % of the lowest. The curves run through each policy's settings, from the one that cleans least, hollow, to the one that cleans most:

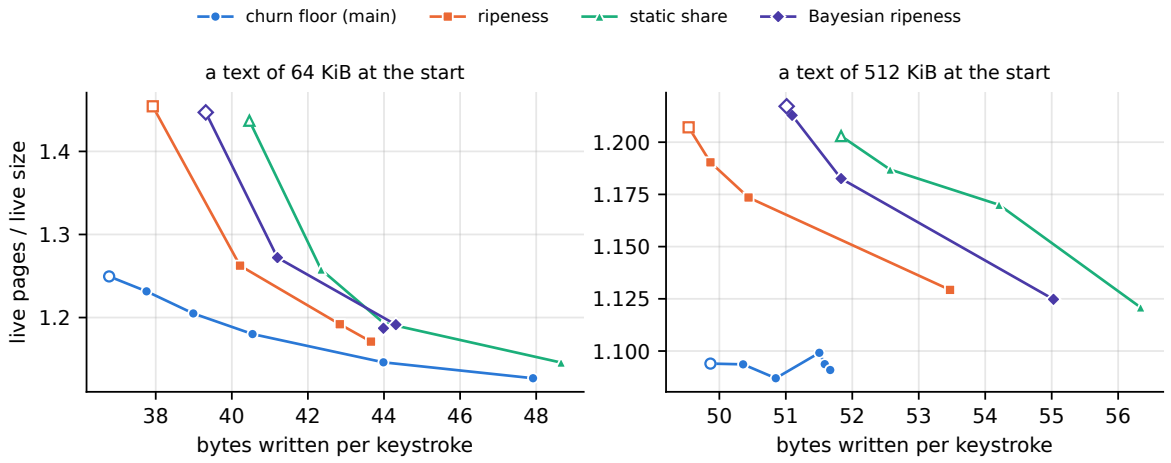


Figure 32.1: Live pages over live size against the bytes written per keystroke, a panel per text, for the churn floor at $\lambda = 1$, 0.5, 0.35, 0.25, 0.15 and 0.1, and the ripeness branches at $\tau = 0.75$, 0.85, 0.9 and 0.95

- **On the texts, the churn floor lies lowest at every setting.** On the 64 KiB text, the ripeness branches keep 2 to 22 % more live pages at equal writes. On the 512 KiB text, every λ keeps 1.09 to 1.10 times the live size at 50 to 52 bytes per keystroke, since the controller holds the budget at one page; the ripeness branches keep 7 to 12 % more live pages at those writes, and come within 4 % of it only at $\tau = 0.95$, with 53.5 to 56.3 bytes per keystroke.
- **The ripeness branches' default target, 0.75, cleans less than main's default on the texts and the world map, and more on the smaller drawings.** It keeps 1.44 and 1.45 times the 64 KiB text's live

size, where main's default keeps 1.25, and 2 to 5 % more live pages than main's default on the world map; on the tiger and the coat of arms, it keeps 2 to 6 % fewer, for 8 to 24 % more writes.

Each drawing takes a row, whose panels share their range of live pages:

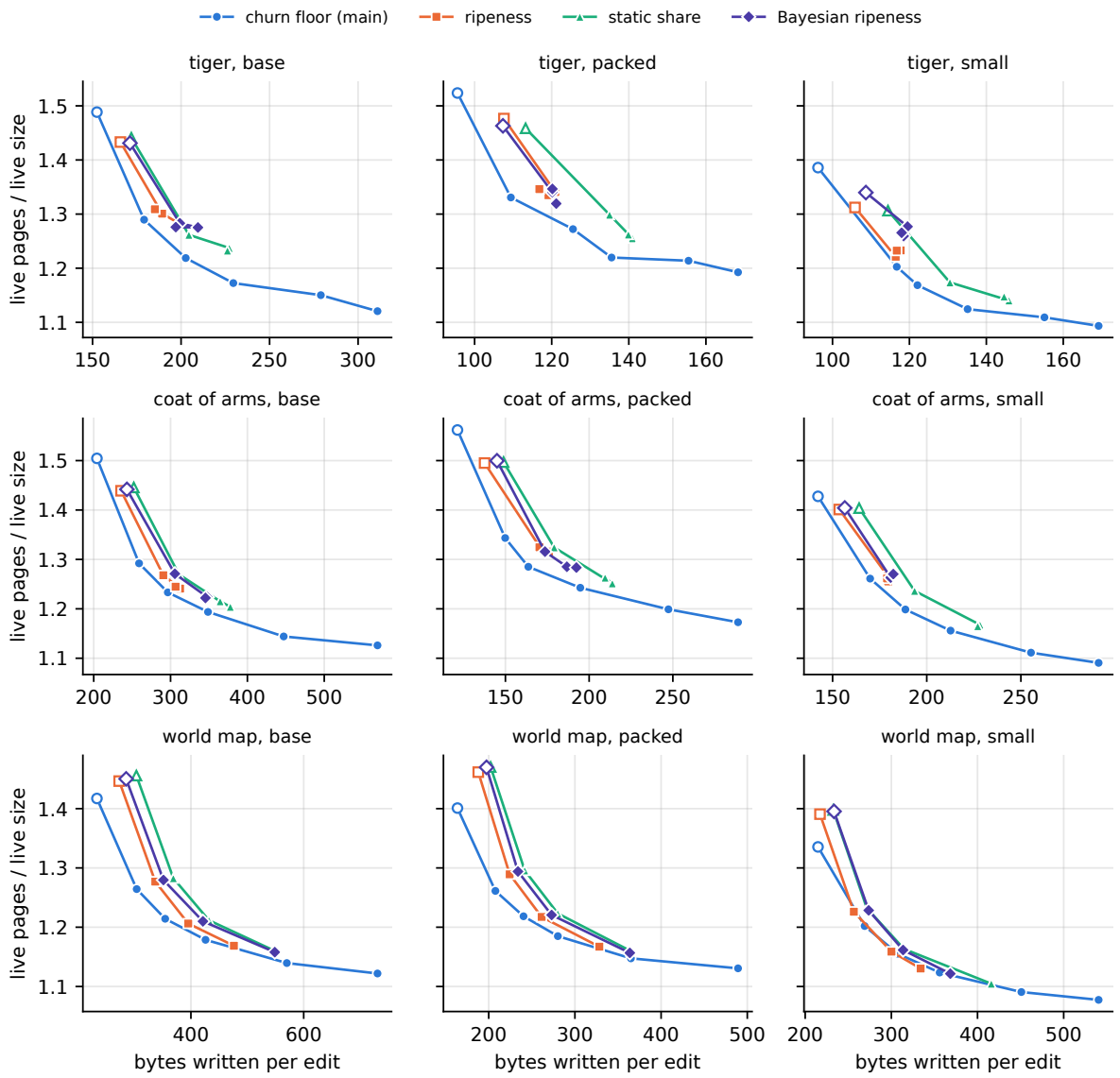


Figure 32.2: Live pages over live size against the bytes written per edit, a panel per drawing and layout, for the churn floor at $\lambda = 1$, 0.5, 0.35, 0.25, 0.15 and 0.1, and the ripeness branches at $\tau = 0.75$, 0.85, 0.9 and 0.95

- **In the drawings' base and packed layouts, the ripeness branches keep 0.3 to 15 % more live pages than the churn floor at equal writes**, the most at $\tau = 0.75$, and the least at their fuller targets on the world map, where single-rate ripeness at $\tau = 0.95$ comes within 0.3 %.
- **In the small layout, the curves come closest**: the ripeness branches keep 1 to 8 % more live pages than the churn floor at equal writes on the tiger and the coat of arms, and on the world map between 0.7 % fewer, single-rate ripeness at $\tau = 0.95$, and 8 % more.
- **On the tiger and the coat of arms, the ripeness branches' curves bunch once their target passes 0.85**: each branch's settings from 0.85 up lie within 0.07 of each other in live pages over live size, at 1.14 to 1.35 times the live size. On the world map, each step of the target still takes live pages off, down to 1.10 to 1.17 times the live size at $\tau = 0.95$.
- **The churn floor reaches further**: at $\lambda = 0.1$ it keeps 1.08 to 1.19 times the drawings' live size, for 168 to 731 bytes per edit, where the best setting of any ripeness branch stops at 1.10 to 1.26.

The equal-write comparisons interpolate the churn floor's curve between its settings, at every setting of the ripeness branches whose writes lie within it.

32.6 Files

Free pages take 14 to 69 % of the drawings' files during the edits, and on the tiger more pages than the live ones under every policy but main's default in the base layout; of the texts' files, they take 11 to 30 %. Each bar is a drawing's file averaged over the second half of the edits, in multiples of the live size:

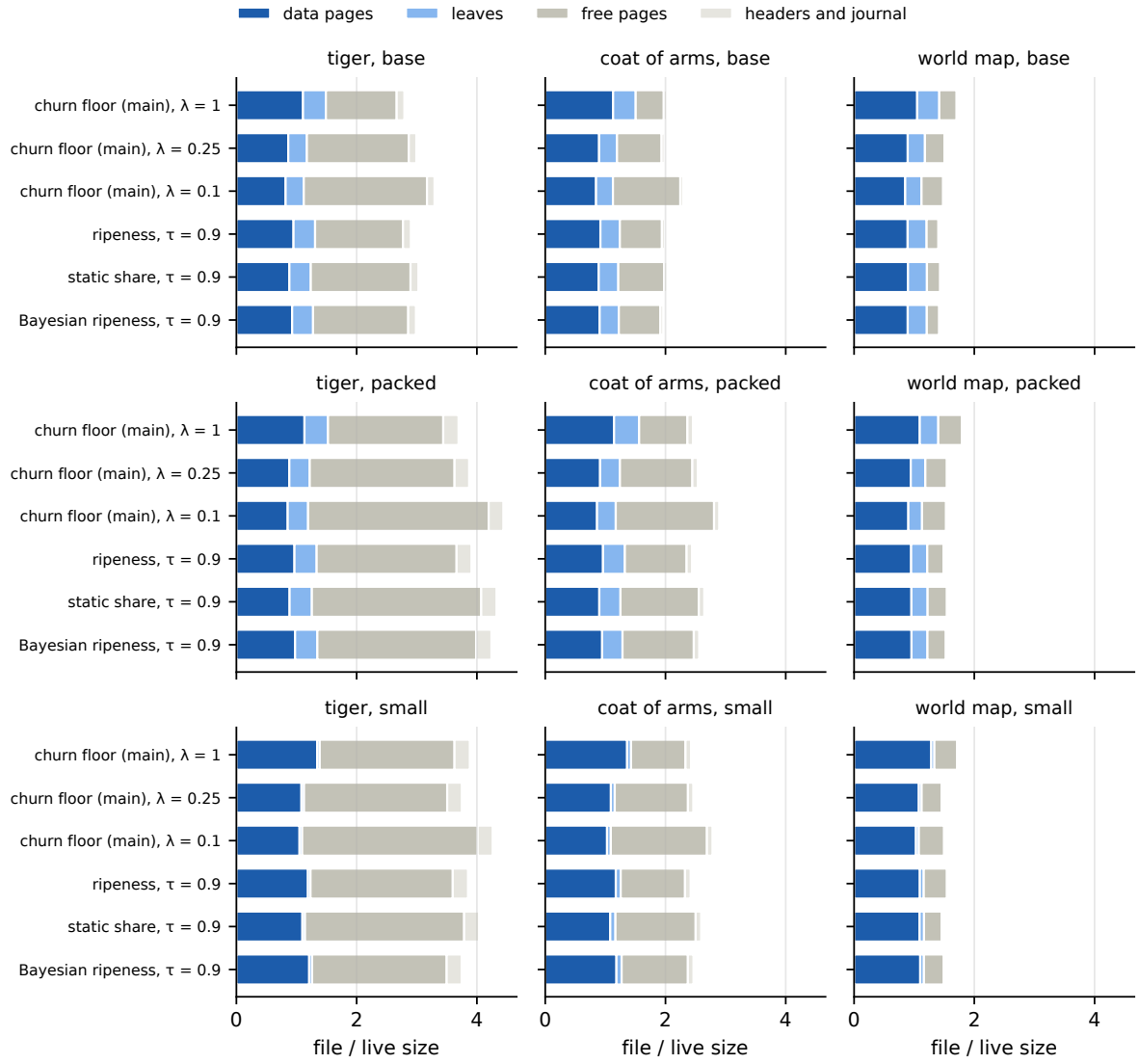


Figure 32.3: What the file holds during the edits, as multiples of the live size: data pages, leaves, free pages, and the headers and the journal, for main's churn floor at $\lambda = 1$, 0.25 and 0.1 and the ripeness branches at $\tau = 0.9$

Each cell gives the file's pages during the edits, and in parentheses the free ones among them:

workload	layout or size	$\lambda = 1$	$\lambda = 0.25$	$\lambda = 0.1$	ripeness	static share	Bayesian
tiger	base	64 (27)	69 (39)	76 (47)	67 (34)	70 (38)	69 (36)
tiger	packed	44 (23)	46 (29)	52 (35)	46 (28)	51 (33)	50 (31)

workload	layout or size	$\lambda = 1$	$\lambda = 0.25$	$\lambda = 0.1$	ripeness	static share	Bayesian
tiger	small	45 (26)	44 (28)	49 (34)	45 (27)	47 (31)	43 (26)
coat of arms	base	127 (29)	125 (47)	144 (71)	125 (44)	128 (48)	124 (44)
coat of arms	packed	80 (26)	83 (39)	94 (53)	80 (33)	86 (43)	84 (39)
coat of arms	small	78 (29)	79 (39)	89 (51)	78 (34)	83 (43)	79 (35)
world map	base	579 (97)	510 (109)	504 (121)	477 (66)	487 (74)	482 (69)
world map	packed	428 (92)	369 (83)	367 (94)	357 (64)	371 (76)	365 (71)
world map	small	408 (91)	346 (78)	355 (98)	367 (91)	345 (68)	354 (77)
text	64 KiB	79 (16)	79 (20)	81 (24)	81 (21)	83 (23)	82 (22)
text	512 KiB	201 (23)	203 (24)	201 (24)	214 (23)	216 (26)	217 (25)

The rest of a file, about three pages, is the two header pages and the journal.

Free pages are returned only in bulk. Truncation waits until 16 pages at the file's end are free, `Options::truncate_tail`, 64 KiB, more than the tiger's live size in its packed and small layouts, and compaction mode starts only once free pages below the tail exceed a quarter of the file, `Options::hole_share`. Cleaning harder frees more pages per flush: at $\lambda = 0.1$, the drawings' files hold 3 to 41 free pages more than at $\lambda = 1$, and compaction mode ran in 2 to 38 flushes where it ran in 1 to 10; the 64 KiB text's file holds 8 more, and the 512 KiB text's, whose budget the controller holds at one page, 1 more. This page has not measured how much of the free space either threshold accounts for.

Closed, the drawings' files take 1.07 to 5.1 times their edited SVG and the texts' 1.24 to 1.71 times the typed text, and which policy closes smallest changes from workload to workload. `close` truncates the file after its last live page, so what it keeps is the free pages below it:

workload	layout or size	edited SVG or text, KiB	$\lambda = 1$	$\lambda = 0.25$	$\lambda = 0.1$	ripeness	static share	Bayesian
tiger	base	58	244	228	296	248	248	228
tiger	packed	58	164	164	172	164	148	148
tiger	small	58	136	164	168	144	156	144
coat of arms	base	180	544	472	524	484	488	480
coat of arms	packed	180	312	312	332	292	272	304
coat of arms	small	180	292	276	324	280	300	296
world map	base	1,045	1,980	1,912	1,796	1,764	1,760	1,756
world map	packed	1,045	1,508	1,356	1,272	1,240	1,188	1,324
world map	small	1,045	1,280	1,120	1,196	1,160	1,156	1,140
text	64 KiB	236	384	360	400	380	404	380
text	512 KiB	684	848	868	876	944	924	932

The closed sizes vary by up to a third between policies that keep nearly as many live pages, since they depend on where the last live page happens to lie when the edits end.

32.7 Writes

Filling the pages costs writes: on the drawings, the churn floor at λ = 0.25 writes 1.4 to 1.8 times as much per edit as at λ = 1, and the ripeness branches at τ = 0.9 1.2 to 1.9 times; on the texts, up to 1.2 times. The journal and the headers write the same under every policy; the rest is data pages and leaves, the flush's own and those cleaning rewrites:

workload	layout or size	journal + headers	λ = 1	λ = 0.25	λ = 0.1	ripeness	static share	Bayesian
tiger	base	52 + 15	152	230	311	185	227	210
tiger	packed	37 + 11	96	136	168	119	141	120
tiger	small	37 + 11	96	135	169	118	146	118
coat of arms	base	65 + 18	204	349	569	312	365	348
coat of arms	packed	44 + 12	121	195	289	170	214	187
coat of arms	small	45 + 13	142	213	291	180	228	182
world map	base	63 + 17	233	426	731	395	432	421
world map	packed	49 + 14	164	280	489	261	280	273
world map	small	51 + 14	215	356	541	300	315	314
text	64 KiB	13.2 + 4.1	36.8	40.5	47.9	42.8	44.0	44.3
text	512 KiB	13.2 + 4.1	49.9	51.5	51.7	50.4	54.2	51.8

All values are bytes per edit on the drawings, and per keystroke on the texts, over the second half of the run. Each cell splits the rest into data pages and leaves:

workload	layout or size	λ = 1	λ = 0.25	λ = 0.1	ripeness	static share	Bayesian
tiger	base	56 + 30	111 + 51	165 + 79	81 + 37	118 + 42	99 + 44
tiger	packed	31 + 16	60 + 27	84 + 35	47 + 23	69 + 23	48 + 23
tiger	small	41 + 7	79 + 8	114 + 8	62 + 7	91 + 7	62 + 8
coat of arms	base	78 + 43	171 + 95	311 + 176	156 + 74	205 + 77	181 + 85
coat of arms	packed	43 + 23	91 + 48	153 + 80	77 + 37	114 + 44	85 + 45
coat of arms	small	63 + 21	129 + 27	208 + 26	97 + 25	146 + 25	101 + 23
world map	base	80 + 73	176 + 169	329 + 322	185 + 130	209 + 143	207 + 134
world map	packed	58 + 42	120 + 97	230 + 196	127 + 70	143 + 73	135 + 75
world map	small	95 + 54	196 + 95	336 + 139	176 + 58	193 + 57	180 + 68
text	64 KiB	4.6 + 14.9	8.0 + 15.2	15.3 + 15.3	11.0 + 14.6	12.6 + 14.1	12.3 + 14.7
text	512 KiB	4.8 + 27.8	4.7 + 29.5	4.8 + 29.5	5.7 + 27.4	9.7 + 27.1	7.4 + 27.1

- **In the drawings' base and packed layouts, the leaves take 25 to 49 % of what goes to data pages and leaves**, more than their 20 to 29 % of the live pages.
- **On the texts, describing the splices writes more than cleaning does.** A keystroke's splice shifts the text behind the cursor, 133 bytes per keystroke on the 64 KiB text and 529 on the 512 KiB one, which the flush states where it lies, restating the statements that describe it. So the leaves take 14 to 15 and 27 to 30 bytes of each keystroke's writes, the journal 13, and the headers 4, about the same under every policy, and the data pages, which cleaning writes, 5 to 15. Splices without restating the tail proposes how to state a splice without that.

Every policy writes far less than saving the file would. Saving the canonical SVG every 100 edits would write 590 bytes per edit on the tiger, 1,845 on the coat of arms, and 10,699 on the world map, and every 1000 edits 59, 185, and 1,070; saving the text every 1000 keystrokes would write 241 and 700 bytes per keystroke. In the regime, kladde writes 118 to 230 bytes per edit on the tiger, 170 to 365 on the coat of arms, and 261 to 432 on the world map, and 40.5 to 54.2 bytes per keystroke on the texts.

32.8 Times

Edits stay fast and flushes short under every policy; on the drawings, the regime costs up to 42 % of the edit rate, and on the world map the static share and the Bayesian posterior take up to 1.7 times single-rate ripeness's flush time; on the texts, the policies differ by little. Each cell gives the edits per second in kladde's calls, in thousands, or on the texts the keystrokes per second, counting the flushes that a full journal starts within an edit; the median time of the flushes at the rows, in milliseconds; and the median time to reopen the closed file, in milliseconds; each the mean of the two timing passes:

workload	layout or size	$\lambda = 1$	$\lambda = 0.25$	$\lambda = 0.1$	ripeness	static share	Bayesian
tiger	base	86 / 3.0 / 1.3	74 / 4.2 / 1.2	68 / 4.8 / 1.1	84 / 3.4 / 1.2	79 / 4.1 / 1.2	73 / 3.6 / 1.2
tiger	packed	119 / 2.8 / 1.2	105 / 3.8 / 1.2	90 / 4.2 / 1.4	107 / 3.4 / 1.3	101 / 4.1 / 1.4	95 / 3.9 / 1.4
tiger	small	111 / 2.6 / 0.8	101 / 3.6 / 0.8	99 / 3.9 / 0.8	108 / 3.2 / 0.8	100 / 3.3 / 0.9	102 / 3.3 / 1.1
coat of arms	base	62 / 3.2 / 3.4	49 / 4.3 / 3.0	34 / 6.9 / 2.9	55 / 3.8 / 3.1	45 / 5.3 / 3.5	42 / 5.3 / 3.3
coat of arms	packed	85 / 3.5 / 3.4	70 / 5.2 / 3.0	58 / 7.5 / 2.9	77 / 4.4 / 3.1	67 / 5.3 / 3.3	60 / 6.5 / 3.0
coat of arms	small	74 / 4.2 / 1.8	67 / 5.0 / 1.7	64 / 5.1 / 1.7	75 / 4.5 / 1.8	64 / 5.3 / 2.0	68 / 4.8 / 1.8
world map	base	37 / 4.5 / 17.6	29 / 6.3 / 15.2	21 / 9.4 / 14.5	31 / 6.6 / 16.0	23 / 8.6 / 17.0	22 / 10.6 / 16.8
world map	packed	45 / 4.3 / 16.3	35 / 5.8 / 14.8	26 / 8.6 / 14.0	39 / 6.0 / 15.4	32 / 7.3 / 16.7	27 / 9.4 / 16.0
world map	small	37 / 6.1 / 8.0	29 / 7.8 / 7.9	24 / 10.8 / 7.6	35 / 6.3 / 8.2	29 / 8.1 / 8.2	26 / 10.5 / 8.2
text	64 KiB	982 / 6.9 / 0.7	1,016 / 6.8 / 0.7	968 / 7.2 / 0.7	990 / 6.8 / 0.7	994 / 7.3 / 0.9	968 / 7.3 / 0.9
text	512 KiB	958 / 9.2 / 1.4	993 / 8.9 / 1.4	966 / 9.0 / 1.5	936 / 9.1 / 1.7	975 / 10.8 / 1.9	970 / 9.4 / 2.2

- **Against main's default, the churn floor at $\lambda = 0.25$ edits the drawings 9 to 23 % slower, and at $\lambda = 0.1$ 11 to 46 %; single-rate ripeness up to 17 %, the static share 8 to 39 %, and the Bayesian posterior 8 to 42 %.** The edit rate falls with the writes, since cleaning works inside the flushes that edits start.
- **The flushes at the rows take 2.6 to 6.1 ms at the median on the drawings under main's default, 3.2 to 10.6 ms in the regime, and 6.8 to 10.8 ms on the texts**, where each row's flush states a thousand

keystrokes' splices, the journal never having filled before it. On the world map, single-rate ripeness flushes as fast as the churn floor at $\lambda = 0.25$ in the base and packed layouts, and faster in the small one; the static share and the Bayesian posterior take 1.2 to 1.7 times as long as single-rate ripeness, more than their 5 to 9 % more writes explain, and on kladde-bench too, their ranking cost more.

- **Reopening the edited file takes 0.8 to 1.4 ms on the tiger, 1.7 to 3.5 ms on the coat of arms, 7.6 to 18 ms on the world map, and 0.7 to 2.2 ms on the texts**; the layout matters more than the policy, since the small layout's address table is small.
- **On the texts, every setting takes 936,000 to 1,016,000 keystrokes per second**, with median flushes of 6.8 to 7.7 ms at 64 KiB and 8.8 to 10.8 ms at 512 KiB; the settings' keystroke rates lie within 9 % of each other, little more than identical runs differ.

Identical runs differed by little. The two timing passes differed in the drawings' edit rate by 1 % at the median and 23 % at most, in their median flush by 3 % and 19 %, and in their reopen times by 2.5 % and 29 %, the largest on the tiger, whose reopens take about a millisecond; on the texts, by 1 to 2 % at the median and at most 8, 10 and 9 %. The reopen times are medians of 21 opens each.

32.9 The sweep in numbers

Every setting's steady state: the live pages over the live size, and the bytes written per edit or keystroke. These are the points of the trade-off figures.

base	tiger	coat of arms	world map
churn floor, $\lambda = 1$	1.49, 152	1.50, 204	1.42, 233
churn floor, $\lambda = 0.5$	1.29, 179	1.29, 259	1.26, 304
churn floor, $\lambda = 0.35$	1.22, 203	1.23, 296	1.21, 354
churn floor, $\lambda = 0.25$	1.17, 230	1.19, 349	1.18, 426
churn floor, $\lambda = 0.15$	1.15, 279	1.14, 447	1.14, 570
churn floor, $\lambda = 0.1$	1.12, 311	1.13, 569	1.12, 731
ripeness, $\tau = 0.75$	1.43, 166	1.44, 236	1.45, 273
ripeness, $\tau = 0.85$	1.30, 189	1.27, 291	1.28, 337
ripeness, $\tau = 0.9$	1.31, 185	1.24, 312	1.21, 395
ripeness, $\tau = 0.95$	1.28, 200	1.24, 307	1.17, 476
static share, $\tau = 0.75$	1.44, 172	1.45, 252	1.46, 303
static share, $\tau = 0.85$	1.26, 204	1.27, 311	1.28, 369
static share, $\tau = 0.9$	1.24, 227	1.22, 365	1.21, 432
static share, $\tau = 0.95$	1.23, 226	1.20, 378	1.16, 548
Bayesian, $\tau = 0.75$	1.43, 171	1.44, 243	1.45, 285
Bayesian, $\tau = 0.85$	1.28, 199	1.27, 306	1.28, 351
Bayesian, $\tau = 0.9$	1.28, 210	1.22, 348	1.21, 421
Bayesian, $\tau = 0.95$	1.28, 197	1.22, 345	1.16, 549

packed	tiger	coat of arms	world map
churn floor, $\lambda = 1$	1.52, 96	1.56, 121	1.40, 164
churn floor, $\lambda = 0.5$	1.33, 109	1.34, 150	1.26, 208
churn floor, $\lambda = 0.35$	1.27, 126	1.29, 164	1.22, 240
churn floor, $\lambda = 0.25$	1.22, 136	1.24, 195	1.18, 280
churn floor, $\lambda = 0.15$	1.21, 155	1.20, 247	1.15, 365
churn floor, $\lambda = 0.1$	1.19, 168	1.17, 289	1.13, 489
ripeness, $\tau = 0.75$	1.48, 108	1.50, 138	1.46, 188
ripeness, $\tau = 0.85$	1.34, 121	1.32, 171	1.29, 224

packed	tiger	coat of arms	world map
ripeness, $\tau = 0.9$	1.33, 119	1.32, 170	1.22, 261
ripeness, $\tau = 0.95$	1.35, 117	1.32, 176	1.17, 328
static share, $\tau = 0.75$	1.46, 113	1.50, 149	1.47, 202
static share, $\tau = 0.85$	1.30, 135	1.32, 179	1.30, 242
static share, $\tau = 0.9$	1.26, 141	1.25, 214	1.22, 280
static share, $\tau = 0.95$	1.26, 140	1.26, 210	1.16, 365
Bayesian, $\tau = 0.75$	1.46, 107	1.50, 145	1.47, 197
Bayesian, $\tau = 0.85$	1.34, 120	1.32, 174	1.29, 234
Bayesian, $\tau = 0.9$	1.35, 120	1.29, 187	1.22, 273
Bayesian, $\tau = 0.95$	1.32, 121	1.28, 192	1.16, 364

small	tiger	coat of arms	world map
churn floor, $\lambda = 1$	1.39, 96	1.43, 142	1.34, 215
churn floor, $\lambda = 0.5$	1.20, 117	1.26, 170	1.20, 269
churn floor, $\lambda = 0.35$	1.17, 122	1.20, 189	1.16, 307
churn floor, $\lambda = 0.25$	1.12, 135	1.16, 213	1.12, 356
churn floor, $\lambda = 0.15$	1.11, 155	1.11, 255	1.09, 451
churn floor, $\lambda = 0.1$	1.09, 169	1.09, 291	1.08, 541
ripeness, $\tau = 0.75$	1.31, 106	1.40, 154	1.39, 217
ripeness, $\tau = 0.85$	1.22, 116	1.27, 179	1.23, 256
ripeness, $\tau = 0.9$	1.23, 118	1.26, 180	1.16, 300
ripeness, $\tau = 0.95$	1.23, 117	1.26, 179	1.13, 334
static share, $\tau = 0.75$	1.31, 114	1.40, 164	1.40, 233
static share, $\tau = 0.85$	1.17, 131	1.24, 194	1.23, 271
static share, $\tau = 0.9$	1.14, 146	1.17, 228	1.16, 315
static share, $\tau = 0.95$	1.15, 145	1.16, 227	1.10, 416
Bayesian, $\tau = 0.75$	1.34, 109	1.40, 156	1.40, 234
Bayesian, $\tau = 0.85$	1.28, 120	1.26, 181	1.23, 274
Bayesian, $\tau = 0.9$	1.26, 118	1.27, 182	1.16, 314
Bayesian, $\tau = 0.95$	1.27, 118	1.27, 182	1.12, 368

text	64 KiB	512 KiB
churn floor, $\lambda = 1$	1.25, 36.8	1.09, 49.9
churn floor, $\lambda = 0.5$	1.23, 37.8	1.09, 50.4
churn floor, $\lambda = 0.35$	1.20, 39.0	1.09, 50.8
churn floor, $\lambda = 0.25$	1.18, 40.5	1.10, 51.5
churn floor, $\lambda = 0.15$	1.15, 44.0	1.09, 51.6
churn floor, $\lambda = 0.1$	1.13, 47.9	1.09, 51.7
ripeness, $\tau = 0.75$	1.45, 37.9	1.21, 49.5
ripeness, $\tau = 0.85$	1.26, 40.2	1.19, 49.9
ripeness, $\tau = 0.9$	1.19, 42.8	1.17, 50.4
ripeness, $\tau = 0.95$	1.17, 43.7	1.13, 53.5
static share, $\tau = 0.75$	1.44, 40.5	1.20, 51.8
static share, $\tau = 0.85$	1.26, 42.3	1.19, 52.6

text	64 KiB	512 KiB
static share, $\tau = 0.9$	1.19, 44.0	1.17, 54.2
static share, $\tau = 0.95$	1.15, 48.7	1.12, 56.3
Bayesian, $\tau = 0.75$	1.45, 39.3	1.22, 51.0
Bayesian, $\tau = 0.85$	1.27, 41.2	1.21, 51.1
Bayesian, $\tau = 0.9$	1.19, 44.3	1.18, 51.8
Bayesian, $\tau = 0.95$	1.19, 44.0	1.12, 55.0

Chapter 33

Ripeness with main's free filling

33.1 Executive summary

With main's free filling, the three ripeness branches write 2 to 3 % less at the same target fill on the drawings and keep about as many live pages, which closes a third to two thirds of their gap to main's churn floor there, and none of it on the texts. At equal writes, the ripeness branches keep, at the median, 2.5 % more live pages than the churn floor in the drawings' base and packed layouts, where they kept 3.7 % more without it, and 1.1 % more in the small layout, where they kept 3.0 % more; on the texts, 8.8 % more, where they kept 8.2 %. Cleaning edited drawings and texts named the free cleaning the ripeness branches give up as the likely source of their extra writes for the same space; it accounts for a third of the gap in the base and packed layouts, two thirds in the small one, and none on the texts.

Free filling takes about one victim per flush on the drawings, as on main, where the draft's took one in seven flushes or fewer, and the branches open fewer pages for cleaning. At $\tau = 0.9$, free filling takes 0.80 to 1.37 victims per flush on the drawings, where the draft's took 0.01 to 0.15 and main's takes 1.0 to 1.3 at $\lambda = 1$, and 0.47 to 0.73 per flush on the texts. The branches evacuate a median 3 % more data pages whole than without it, 9 to 25 % of them free where 0 to 3 % were, and open 2 to 20 % fewer pages for cleaning, 7 % fewer at the median.

At the same setting, the drawings' writes fall by a median 3.3 % in the base and packed layouts and 2.7 % in the small one, in 98 of 108 runs, while the live pages stay within 2.5 %; on the texts, writes fall by 0.4 % at the median, and both move in either direction, the most at $\tau = 0.75$. Along each branch's curve, the drawings keep a median 1.2 % fewer live pages at equal writes in the base and packed layouts, and 1.1 % fewer in the small one, the most at $\tau = 0.75$, up to 5.7 % fewer, and about 0.3 % fewer at $\tau = 0.9$ and 0.95.

Single-rate ripeness gains most, and lies below the churn floor at 8 of its settings in the small layout, by up to 2.4 %; the churn floor still lies lowest nearly everywhere else, and reaches further. At equal writes, single-rate ripeness keeps a median 1.9 % more live pages than the churn floor in the drawings' base and packed layouts, where it kept 3.6 % more, 0.5 % fewer in the small layout, where it kept 2.3 % more, and 3.3 % more on the texts, where it kept 7.0 % more. The static share and the Bayesian posterior move less, and on the texts by less than a point, in either direction. The fullest any branch fills the pages moves by 0.02 times the live size at most: at $\lambda = 0.1$, the churn floor keeps 1.08 to 1.19 times the drawings' live size, and the best setting of any branch 1.10 to 1.25.

Otherwise little changes: the ripeness branches' pages fill as before, 80 to 88 % on the tiger and 84 to 90 % on the coat of arms at $\tau = 0.9$, and their leaves as before, 78 to 86 % in the drawings' base and packed layouts.

33.2 What changed

Free filling on the ripeness branches is main's. The room left in a page the flush writes anyway takes, in compaction mode, the tail first, then whole data victims that fit, the sparsest first, then description defragmentation's candidates beyond its reserved share, then as many of one victim's statements as fit,

whether ripe or not. Without it, the branches gave that room to ripe victims whose survivors take at most θ, 5 %, of a page, and then to the cursor, the least-filled data page of the previous flush, a whole statement at a time, as the ripeness draft did then; both are gone, and the draft takes main's free filling too.

The price of space holds after a flush whose only cleaning was free filling. The controller holds the price where moving it could change nothing: it does not lower it after a flush that cleaned nothing. Free filling takes its victims whatever the price, in nearly every flush, so counted as cleaning, it would let the price fall in every flush while the pages are fuller than the target; its victims do not count.

branch	without main's free filling	with it
`ripeness-svg`	`d6c444d`	`3a4209a`
`ripeness2-svg`	`37a3a22`	`59467ed`
`bayesian-ripeness-svg`	`d11b834`	`1055cb3`

The commits without main's free filling are not those Cleaning edited drawings and texts measured. Each branch has a commit since that fixes what its consolidator state records: the estimates the state's own rewrite changes, after its record is written, were never recorded, and came back at a reopen as losses in the gap. That changes only bytes the consolidator state writes, and nothing on the texts, but on the drawings it moves single runs by up to 3.1 % in live pages and 7.3 % in writes, where the ranking amplifies a small difference in what a flush writes into a different choice of victims. So this page compares against its own runs of those commits, which the earlier page's tables come within those bounds of, and the spread of single runs is part of every change below: a change in one run smaller than a few percent says little, while a change of one sign in most runs does.

33.3 What was measured

Everything ran as on the earlier page: the same drawings, layouts, edits, texts, keystrokes and settings, the three ripeness branches at $\tau = 0.75$, 0.85, 0.9 and 0.95, on the same host, with `text-bench` from kladders `7c5cdd6` added to every commit, without its `--churn-floor` option. The churn floor's numbers are the earlier page's, from its data in `data/cleaning-edits/`; main's free filling is the churn floor's own, so its runs would not change. This page ran 264 runs, eight at a time, 132 of each commit, and no timing passes, so it reports no times. `data/ripeness-free-filling/` keeps every run with main's free filling, and its `before/` those without, each with its table and, under `logs/`, the end of its output.

To reproduce a run, build as on the earlier page, on the commit with main's free filling:

```
git cherry-pick --no-commit 7c5cdd6   # text-bench; drop its --churn-floor option
cargo build --release -p kladde-svg --bin svg-bench --features packed
target/release/svg-bench crates/kladde-svg/corpus/drawings/world-map.svg \
    out/ --edits 177831 --journal-budget 4 --target-fill 0.9
cargo build --release -p kladde-bench --bin text-bench
target/release/text-bench out/ --size 524288 --journal-budget 4 --target-fill 0.9
```

The figures come from the tables here, those without main's free filling labelled `before/…`, and the churn floor's from the earlier page's:

```
D=content/evaluation/data; F=content/evaluation/figures/ripeness-free-filling
runs=()
for dir in $D/ripeness-free-filling/*-*-*/ $D/ripeness-free-filling/before/*-*-*/ \
        $D/cleaning-edits/main-*-*/; do
    IFS=- read -r p k l <<< "$(basename $dir)"; knob=${k%%[0-9]*}
    case $dir in */before/*) before=before/ ;; *) before= ;; esac
    runs+=("$before$p/$knob/${k#$knob}/$l=$dir")
done
tools/plot-evaluation.py --only svg-tradeoff,text-tradeoff --out $F "${runs[@]}"
```

The tables give the churn floor at $\lambda = 0.25$ for reference, and each ripeness branch at $\tau = 0.9$, the earlier page's regime, with its value without main's free filling in brackets.

33.4 Free filling

Free filling takes 69 to 895 victims per run on the drawings at τ = 0.9, where the draft's took 1 to 98, and as many as main's at λ = 1 to within 30 %; at that target, the branches open fewer pages for cleaning in every run. Each cell gives, over a whole run, the data victims that free filling took and the data pages opened only to take victims:

workload	layout or size	churn floor, λ = 0.25	ripeness	static share	Bayesian
tiger	base	66 / 545	96 / 290 [3 / 357]	107 / 511 [5 / 555]	104 / 374 [10 / 381]
tiger	packed	56 / 301	77 / 170 [8 / 196]	69 / 311 [1 / 336]	77 / 176 [1 / 221]
tiger	small	66 / 413	76 / 216 [8 / 241]	72 / 398 [7 / 418]	70 / 228 [5 / 241]
coat of arms	base	131 / 1,045	155 / 788 [19 / 838]	165 / 1,096 [19 / 1,184]	160 / 914 [18 / 979]
coat of arms	packed	90 / 604	118 / 396 [11 / 457]	108 / 661 [16 / 672]	121 / 451 [13 / 511]
coat of arms	small	98 / 883	116 / 493 [11 / 536]	105 / 846 [8 / 945]	112 / 513 [6 / 580]
world map	base	673 / 4,729	773 / 4,423 [75 / 4,494]	863 / 4,421 [88 / 4,635]	816 / 4,455 [86 / 4,656]
world map	packed	573 / 3,130	670 / 2,876 [61 / 3,121]	890 / 3,000 [93 / 3,324]	780 / 2,867 [69 / 3,205]
world map	small	571 / 5,996	790 / 4,327 [80 / 4,420]	895 / 4,640 [98 / 4,877]	849 / 4,211 [84 / 4,368]
text	64 KiB	50 / 69	47 / 93 [8 / 117]	48 / 132 [8 / 148]	52 / 118 [4 / 133]
text	512 KiB	57 / 101	63 / 126 [6 / 137]	67 / 171 [9 / 178]	73 / 142 [6 / 146]

- **The branches evacuate about as many data pages whole as before, and more of them free.** At τ = 0.9, they evacuate between 3 % fewer and 11 % more on the drawings than without main's free filling, 3 % more at the median, and 9 to 25 % of them by free filling, where 0 to 3 % were; on the texts, 7 to 33 % more, 24 to 33 % of them free.
- **So they open fewer pages for cleaning**: 2 to 20 % fewer on the drawings at τ = 0.9, 7 % at the median, and 3 to 21 % on the texts. Over every setting, as many as 38 % fewer on the drawings and 68 % on the 64 KiB text, both at τ = 0.75.
- **Neither count sees what the cursor cleaned without main's free filling**: it took the room that small ripe victims left, a whole statement at a time, and freed its page only once the page was empty, which counts as no victim taken.

33.5 The trade-off between space and writes

Main's free filling moves each branch's curve down and to the left on the drawings, the most at τ = 0.75 and by a fraction of a percent at τ = 0.9 and 0.95, and closes part of the gap to the churn floor; on the texts, it moves the curves by about as much in either direction. Each panel shows each policy's settings, from the one that cleans least, hollow, to the one that cleans most, and dashed and pale each ripeness branch's curve without main's free filling; each drawing takes a row, whose panels share their range of live pages:

- **At equal writes, each branch keeps between 5.7 % fewer and 0.1 % more live pages than without main's free filling, 1.2 % fewer at the median in the base and packed layouts and 1.1 % fewer in**

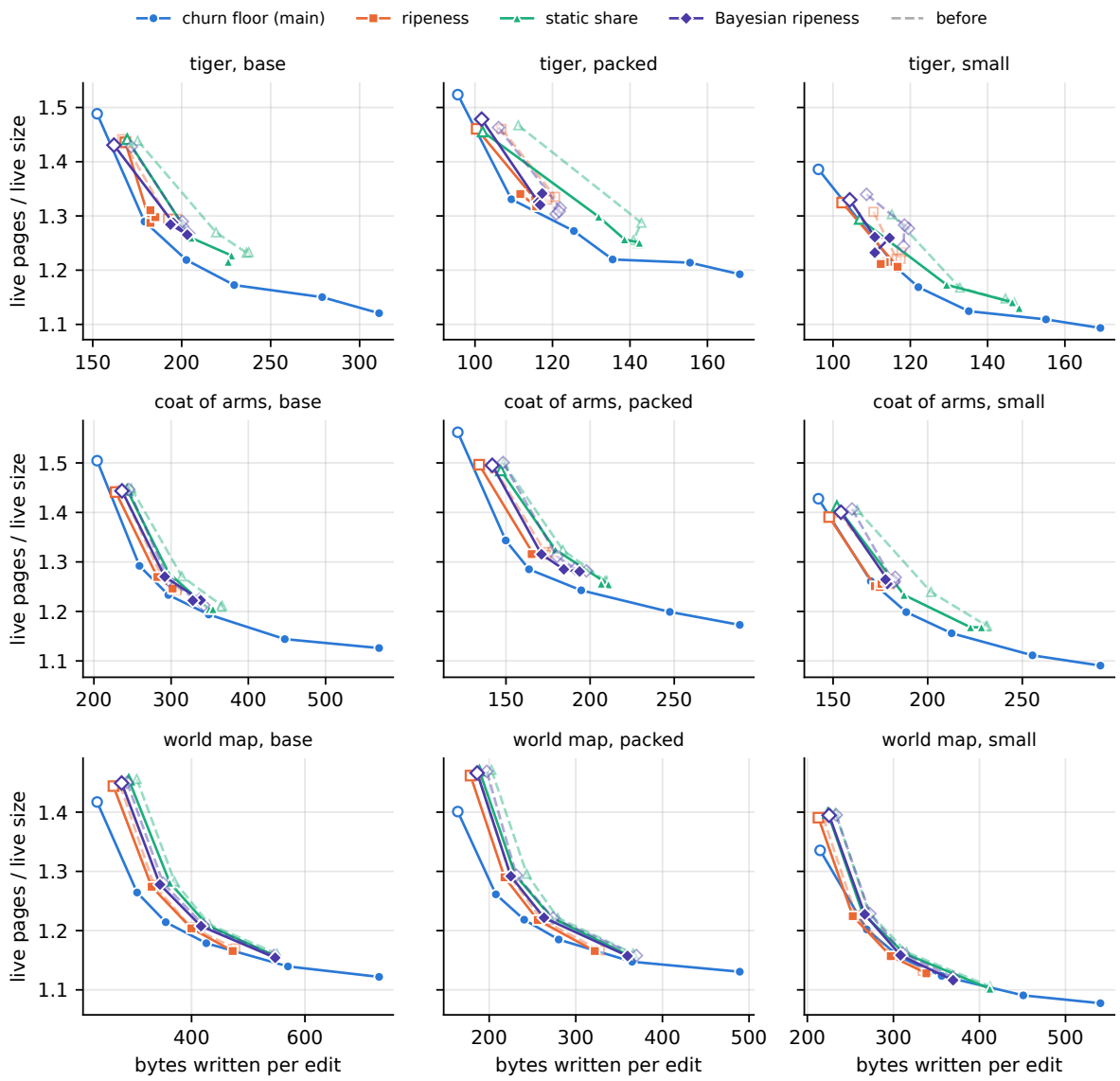


Figure 33.1: Live pages over live size against the bytes written per edit, a panel per drawing and layout, for the churn floor at $\lambda = 1$, 0.5, 0.35, 0.25, 0.15 and 0.1, and the ripeness branches at $\tau = 0.75$, 0.85, 0.9 and 0.95, with and, dashed, without main's free filling

the small one, at 42 of 44 and 22 of 22 settings whose writes the other curve spans. At $\tau = 0.75$, 2.4 and 2.9 % fewer at the median; at $\tau = 0.9$ and 0.95, 0.3 % fewer.

- **Against the churn floor's curve, the branches keep between 0.1 % fewer and 12 % more live pages at equal writes in the base and packed layouts, where they kept between 0.2 and 15 % more, and between 2.4 % fewer and 6.5 % more in the small layout, where they kept between 0.6 % fewer and 8.3 % more.** At the median, 2.5 % more where they kept 3.7 % more, and 1.1 % more where they kept 3.0 %. 13 of their 107 settings in range lie below the churn floor's curve, where 3 did: 10 of single-rate ripeness's, one of the static share's and two of the Bayesian posterior's, all but two in the small layout, and by 2.4 % at the most, single-rate ripeness at $\tau = 0.95$ on the tiger; the two in the world map's base and packed layouts lie below by 0.1 % or less.
- **Single-rate ripeness gains most.** At the median, it keeps 1.9 % more live pages than the churn floor at equal writes in the base and packed layouts, where it kept 3.6 % more, and 0.5 % fewer in the small layout, where it kept 2.3 % more; the static share 3.4 and 2.1 % more, where it kept 5.3 and 2.9 %, and the Bayesian posterior 2.9 and 1.8 % more, where it kept 3.6 and 4.2 %.
- **The furthest each branch reaches stays where it was**, within 0.02 times the live size.

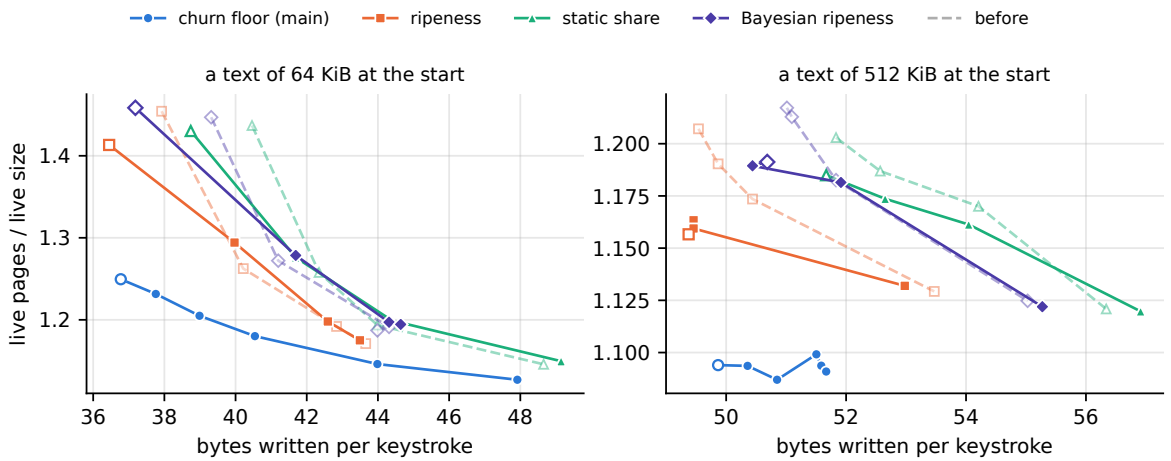


Figure 33.2: Live pages over live size against the bytes written per keystroke, a panel per text, for the churn floor at $\lambda = 1$, 0.5, 0.35, 0.25, 0.15 and 0.1, and the ripeness branches at $\tau = 0.75$, 0.85, 0.9 and 0.95, with and, dashed, without main's free filling

- **On the texts, the gap to the churn floor stays where it was**: at equal writes, the branches keep between 2.1 and 18 % more live pages than the churn floor, where they kept between 1.9 and 22 % more, 8.8 % at the median, where they kept 8.2 %.
- **On the 512 KiB text, single-rate ripeness at $\tau = 0.75$ to 0.9 keeps 1.16 times the live size, where it kept 1.17 to 1.21**, for about as many writes, 49.4 to 49.5 bytes per keystroke; at $\tau = 0.95$, it keeps 1.13 times, as before. The static share and the Bayesian posterior keep 1 to 2 % fewer live pages at $\tau = 0.75$ and 0.85, and within 1 % of as many at $\tau = 0.9$ and 0.95.
- **On the 64 KiB text, the branches at $\tau = 0.75$ write 4 to 5 % less**, single-rate ripeness for 3 % fewer live pages and the other two for about as many; at the other targets, live pages and writes change by 2.5 % at most, in either direction.

33.6 The regime

For the space each branch kept at $\tau = 0.9$ without main's free filling, it writes between 9 % less and 2 % more with it, and on the drawings 0 to 9 % less. Each cell gives the live pages over the live size each branch reached at $\tau = 0.9$ without main's free filling, the bytes written per edit or keystroke then, and the bytes it writes for that space with main's free filling, interpolated along its curve:

workload	layout or size	ripeness, $\tau = 0.9$	static share, $\tau = 0.9$	Bayesian, $\tau = 0.9$
tiger	base	1.29: 192 → 182 (−5 %)	1.23: 236 → 218 (−8 %)	1.28: 200 → 197 (−1 %)
tiger	packed	1.33: 119 → 114 (−4 %)	1.26: 141 → 139 (−1 %)	1.30: 121 → —
tiger	small	1.23: 118 → 110 (−6 %)	1.14: 147 → 146 (0 %)	1.28: 118 → 107 (−9 %)
coat of arms	base	1.24: 307 → —	1.21: 366 → 344 (−6 %)	1.21: 342 → —
coat of arms	packed	1.32: 178 → 165 (−7 %)	1.26: 209 → 204 (−2 %)	1.29: 191 → 184 (−4 %)
coat of arms	small	1.26: 180 → 171 (−5 %)	1.17: 231 → 221 (−4 %)	1.26: 182 → 179 (−2 %)
world map	base	1.21: 399 → 398 (0 %)	1.21: 431 → 427 (−1 %)	1.21: 421 → 415 (−1 %)
world map	packed	1.22: 261 → 256 (−2 %)	1.22: 279 → 273 (−2 %)	1.22: 274 → 264 (−4 %)
world map	small	1.16: 300 → 296 (−1 %)	1.16: 315 → 311 (−1 %)	1.16: 311 → 306 (−1 %)
text	64 KiB	1.19: 42.8 → 42.8 (0 %)	1.19: 44.0 → 44.9 (+2 %)	1.19: 44.3 → —
text	512 KiB	1.17: 50.4 → ≤ 49.4 (−2 %)	1.17: 54.2 → 53.1 (−2 %)	1.18: 51.8 → 51.8 (0 %)

A dash means that no setting with main's free filling reaches that space; the nearest misses it by at most 1.3 %. On the 512 KiB text, single-rate ripeness with main's free filling keeps less space even at $\tau = 0.75$, its setting that writes least.

33.7 Live pages and fill

At $\tau = 0.9$, each branch keeps between 2 % fewer and 1 % more live pages than without main's free filling, and fills its pages as before. Each cell gives the live pages, the leaves among them in parentheses, and in brackets the change from the branch without main's free filling; the first column gives the live bytes they hold, in data pages and in leaves:

workload	layout or size	live bytes, KiB	churn floor, $\lambda = 0.25$	ripeness	static share	Bayesian
tiger	base	72 + 26	26.9 (7.1)	29.8 (8.1) [0 %]	28.2 (8.1) [0 %]	29.1 (7.8) [−1 %]
tiger	packed	38 + 14	14.4 (4.0)	15.6 (4.2) [−1 %]	14.9 (4.4) [0 %]	15.6 (4.3) [+1 %]
tiger	small	46 + 1	13.1 (0.5)	14.1 (0.4) [−2 %]	13.2 (0.5) [0 %]	14.6 (0.5) [−2 %]
coat of arms	base	205 + 67	75.2 (18.8)	78.4 (20.4) [0 %]	75.9 (20.1) [−1 %]	77.0 (19.8) [+1 %]
coat of arms	packed	108 + 38	40.6 (10.8)	43.0 (11.7) [0 %]	41.0 (11.4) [−1 %]	42.0 (11.1) [0 %]
coat of arms	small	128 + 7	37.1 (2.0)	40.1 (2.5) [0 %]	37.5 (2.6) [0 %]	40.3 (2.7) [0 %]

workload	layout or size	live bytes, KiB	churn floor, $\lambda = 0.25$	ripeness	static share	Bayesian
world map	base	1,117 + 343	398.6 (96.6)	407.2 (104.2) [0 %]	408.8 (104.6) [0 %]	408.4 (105.3) [0 %]
world map	packed	838 + 202	282.2 (57.0)	290.1 (63.1) [0 %]	291.1 (63.5) [0 %]	291.1 (62.8) [0 %]
world map	small	939 + 38	264.6 (10.6)	272.6 (14.6) [0 %]	273.7 (16.8) [0 %]	272.9 (13.5) [0 %]
text	64 KiB	192 + 13	56.9 (3.8)	57.7 (3.8) [0 %]	57.7 (4.3) [0 %]	57.6 (3.8) [0 %]
text	512 KiB	641 + 27	176.5 (7.6)	186.3 (10.8) [−1 %]	186.5 (11.4) [−1 %]	189.8 (10.2) [0 %]

Over the live size, with the value without main's free filling in brackets:

workload	layout or size	churn floor, $\lambda = 0.25$	ripeness	static share	Bayesian
tiger	base	1.17	1.30 [1.29]	1.23 [1.23]	1.27 [1.28]
tiger	packed	1.22	1.32 [1.33]	1.26 [1.26]	1.32 [1.30]
tiger	small	1.12	1.21 [1.23]	1.14 [1.14]	1.26 [1.28]
coat of arms	base	1.19	1.24 [1.24]	1.20 [1.21]	1.22 [1.21]
coat of arms	packed	1.24	1.32 [1.32]	1.25 [1.26]	1.29 [1.29]
coat of arms	small	1.16	1.25 [1.26]	1.17 [1.17]	1.26 [1.26]
world map	base	1.18	1.20 [1.21]	1.21 [1.21]	1.21 [1.21]
world map	packed	1.18	1.22 [1.22]	1.22 [1.22]	1.22 [1.22]
world map	small	1.12	1.16 [1.16]	1.16 [1.16]	1.16 [1.16]
text	64 KiB	1.18	1.20 [1.19]	1.20 [1.19]	1.20 [1.19]
text	512 KiB	1.10	1.16 [1.17]	1.16 [1.17]	1.18 [1.18]

Each cell gives the fill of the data pages, of the leaves, and of all live pages, in percent, with the values without main's free filling in brackets:

workload	layout or size	churn floor, $\lambda = 0.25$	ripeness	static share	Bayesian
tiger	base	91 / 90 / 91	83 / 79 / 82 [83 / 81 / 83]	90 / 80 / 87 [90 / 79 / 87]	85 / 83 / 85 [85 / 80 / 84]
tiger	packed	89 / 88 / 89	83 / 81 / 82 [82 / 80 / 81]	90 / 78 / 87 [90 / 79 / 87]	84 / 80 / 83 [84 / 83 / 83]
tiger	small	91 / 33 / 89	84 / 29 / 82 [83 / 28 / 81]	90 / 30 / 88 [90 / 29 / 88]	81 / 29 / 80 [80 / 30 / 79]
coat of arms	base	90 / 89 / 90	88 / 82 / 87 [89 / 82 / 87]	92 / 83 / 90 [92 / 83 / 90]	90 / 85 / 89 [91 / 85 / 89]
coat of arms	packed	90 / 89 / 90	86 / 82 / 85 [86 / 82 / 85]	92 / 83 / 90 [92 / 82 / 89]	88 / 86 / 87 [88 / 86 / 87]
coat of arms	small	91 / 81 / 90	85 / 66 / 84 [85 / 64 / 84]	92 / 66 / 90 [92 / 64 / 90]	85 / 63 / 84 [85 / 61 / 83]
world map	base	92 / 89 / 92	93 / 83 / 90 [93 / 83 / 90]	92 / 83 / 90 [92 / 83 / 90]	93 / 82 / 90 [93 / 82 / 90]

workload	layout or size	churn floor, λ = 0.25	ripeness	static share	Bayesian
world map	packed	93 / 89 / 92	93 / 80 / 90 [93 / 81 / 90]	93 / 80 / 90 [93 / 81 / 90]	92 / 81 / 90 [92 / 81 / 90]
world map	small	93 / 91 / 92	91 / 65 / 90 [91 / 67 / 90]	92 / 58 / 90 [92 / 59 / 90]	91 / 70 / 90 [91 / 70 / 90]
text	64 KiB	91 / 87 / 90	89 / 83 / 89 [90 / 81 / 90]	90 / 75 / 89 [91 / 74 / 89]	90 / 84 / 89 [90 / 84 / 90]
text	512 KiB	95 / 93 / 95	92 / 63 / 90 [90 / 65 / 89]	92 / 61 / 90 [91 / 61 / 89]	90 / 67 / 88 [90 / 65 / 88]

- **On the tiger and the coat of arms, the branches still stop short of their target**, their live pages 80 to 88 % and 84 to 90 % full at $\tau = 0.9$, as without main's free filling; on the world map they reach it, at 90 %, as before.
- **The leaves fill as before**: 78 to 86 % in the drawings' base and packed layouts, against 79 to 86 %, and the branches rewrite about as many of them.

33.8 Files

The files hold between 7 % fewer and 4 % more pages than without main's free filling, and close between 20 % smaller and 15 % larger, by about as much as runs that keep as many live pages differ. Each cell gives the file's pages during the edits, and in parentheses the free ones among them, with the values without main's free filling in brackets:

workload	layout or size	churn floor, λ = 0.25	ripeness	static share	Bayesian
tiger	base	69 (39)	67 (35) [65 (32)]	69 (38) [71 (40)]	67 (34) [66 (34)]
tiger	packed	46 (29)	48 (29) [46 (28)]	48 (30) [51 (33)]	48 (29) [48 (30)]
tiger	small	44 (28)	43 (26) [45 (27)]	47 (31) [47 (31)]	44 (26) [47 (29)]
coat of arms	base	125 (47)	124 (43) [123 (41)]	126 (47) [128 (48)]	124 (44) [122 (43)]
coat of arms	packed	83 (39)	80 (34) [82 (36)]	85 (41) [84 (40)]	83 (38) [84 (39)]
coat of arms	small	79 (39)	75 (32) [78 (34)]	81 (41) [82 (41)]	77 (34) [80 (37)]
world map	base	510 (109)	477 (67) [477 (66)]	483 (72) [483 (70)]	481 (69) [500 (88)]
world map	packed	369 (83)	358 (65) [357 (64)]	376 (82) [373 (79)]	354 (60) [358 (64)]
world map	small	346 (78)	347 (71) [343 (68)]	347 (70) [351 (74)]	339 (63) [347 (70)]
text	64 KiB	79 (20)	82 (22) [81 (21)]	82 (22) [83 (23)]	82 (22) [82 (22)]
text	512 KiB	203 (24)	211 (23) [214 (23)]	215 (26) [216 (26)]	216 (24) [217 (25)]

Each cell gives the closed file in KiB, with the value without main's free filling in brackets:

workload	layout or size	churn floor, λ = 0.25	ripeness	static share	Bayesian
tiger	base	228	216 [220]	264 [252]	232 [236]
tiger	packed	164	144 [164]	164 [148]	140 [176]
tiger	small	164	152 [144]	152 [156]	140 [128]
coat of arms	base	472	476 [496]	500 [504]	464 [476]
coat of arms	packed	312	292 [320]	320 [308]	280 [276]
coat of arms	small	276	260 [280]	312 [272]	284 [248]
world map	base	1,912	1,788 [1,756]	1,796 [1,780]	1,744 [1,956]
world map	packed	1,356	1,224 [1,200]	1,208 [1,364]	1,188 [1,304]
world map	small	1,120	1,156 [1,176]	1,180 [1,200]	1,120 [1,168]
text	64 KiB	360	364 [380]	356 [404]	360 [380]
text	512 KiB	868	940 [944]	928 [924]	956 [932]

As on the earlier page, a closed size depends on where the last live page lies when the edits end.

33.9 Writes

At τ = 0.9, each branch writes between 5 % less and 1 % more than without main's free filling, nearly all of the difference in data pages. Each cell gives the bytes written per edit, or per keystroke on the texts, with the change from the branch without main's free filling in brackets; the journal and the headers write what they wrote before, as on the earlier page:

workload	layout or size	churn floor, λ = 0.25	ripeness	static share	Bayesian
tiger	base	230	185 [−4 %]	228 [−4 %]	203 [+1 %]
tiger	packed	136	116 [−3 %]	139 [−2 %]	117 [−3 %]
tiger	small	135	113 [−4 %]	146 [0 %]	115 [−3 %]
coat of arms	base	349	304 [−1 %]	349 [−5 %]	338 [−1 %]
coat of arms	packed	195	168 [−5 %]	211 [+1 %]	184 [−3 %]
coat of arms	small	213	175 [−3 %]	223 [−4 %]	179 [−2 %]
world map	base	426	400 [0 %]	429 [−1 %]	417 [−1 %]
world map	packed	280	256 [−2 %]	272 [−3 %]	263 [−4 %]
world map	small	356	297 [−1 %]	313 [−1 %]	308 [−1 %]
text	64 KiB	40.5	42.6 [−1 %]	44.4 [+1 %]	44.3 [0 %]
text	512 KiB	51.5	49.5 [−2 %]	54.0 [0 %]	51.9 [0 %]

Each cell splits the rest into data pages and leaves, with the split without main's free filling in brackets:

workload	layout or size	churn floor, λ = 0.25	ripeness	static share	Bayesian
tiger	base	111 + 51	79 + 39 [86 + 39]	117 + 44 [123 + 46]	92 + 44 [93 + 40]
tiger	packed	60 + 27	46 + 22 [47 + 23]	67 + 23 [69 + 23]	46 + 23 [49 + 23]
tiger	small	79 + 8	59 + 7 [62 + 7]	91 + 7 [91 + 8]	59 + 8 [62 + 8]
coat of arms	base	171 + 95	150 + 72 [153 + 72]	192 + 75 [203 + 80]	172 + 83 [177 + 83]

workload	layout or size	churn floor, λ = 0.25	ripeness	static share	Bayesian
coat of arms	packed	91 + 48	74 + 38 [82 + 40]	112 + 43 [110 + 43]	82 + 46 [88 + 47]
coat of arms	small	129 + 27	95 + 23 [97 + 25]	141 + 25 [149 + 25]	97 + 24 [101 + 24]
world map	base	176 + 169	190 + 130 [188 + 131]	208 + 140 [209 + 142]	203 + 133 [205 + 135]
world map	packed	120 + 97	123 + 70 [128 + 69]	136 + 72 [142 + 73]	127 + 72 [136 + 75]
world map	small	196 + 95	175 + 57 [177 + 57]	192 + 55 [193 + 56]	179 + 64 [179 + 67]
text	64 KiB	8.0 + 15.2	10.8 + 14.5 [11.0 + 14.6]	12.3 + 14.8 [12.6 + 14.1]	12.4 + 14.7 [12.3 + 14.7]
text	512 KiB	4.7 + 29.5	5.2 + 27.0 [5.7 + 27.4]	9.4 + 27.3 [9.7 + 27.1]	7.8 + 26.8 [7.4 + 27.1]

On the drawings, the data pages take up to 11 bytes less per edit, and the leaves between 6 bytes less and 4 more.

33.10 The sweep in numbers

Every setting's steady state: the live pages over the live size, and the bytes written per edit or keystroke, with the values without main's free filling in brackets. These are the points of the trade-off figures; the churn floor's are on the earlier page.

base	tiger	coat of arms	world map
ripeness, τ = 0.75	1.44, 168 [1.44, 167]	1.44, 228 [1.44, 236]	1.44, 262 [1.45, 274]
ripeness, τ = 0.85	1.29, 182 [1.29, 192]	1.27, 282 [1.27, 288]	1.27, 330 [1.28, 333]
ripeness, τ = 0.9	1.30, 185 [1.29, 192]	1.24, 304 [1.24, 307]	1.20, 400 [1.21, 399]
ripeness, τ = 0.95	1.31, 182 [1.29, 197]	1.25, 302 [1.25, 307]	1.17, 473 [1.17, 477]
static share, τ = 0.75	1.44, 169 [1.44, 175]	1.45, 243 [1.45, 248]	1.45, 289 [1.46, 303]
static share, τ = 0.85	1.26, 205 [1.27, 219]	1.28, 297 [1.27, 314]	1.28, 362 [1.28, 370]
static share, τ = 0.9	1.23, 228 [1.23, 236]	1.20, 349 [1.21, 366]	1.21, 429 [1.21, 431]
static share, τ = 0.95	1.22, 226 [1.23, 238]	1.20, 354 [1.21, 364]	1.16, 550 [1.16, 551]
Bayesian, τ = 0.75	1.43, 162 [1.43, 172]	1.44, 236 [1.45, 245]	1.45, 277 [1.45, 286]
Bayesian, τ = 0.85	1.29, 195 [1.27, 204]	1.27, 292 [1.26, 297]	1.28, 344 [1.28, 350]
Bayesian, τ = 0.9	1.27, 203 [1.28, 200]	1.22, 338 [1.21, 342]	1.21, 417 [1.21, 421]
Bayesian, τ = 0.95	1.28, 194 [1.29, 200]	1.22, 328 [1.22, 337]	1.15, 548 [1.16, 546]

packed	tiger	coat of arms	world map
ripeness, τ = 0.75	1.46, 100 [1.46, 107]	1.50, 134 [1.49, 143]	1.46, 178 [1.46, 187]
ripeness, τ = 0.85	1.33, 116 [1.34, 121]	1.31, 167 [1.32, 173]	1.29, 218 [1.29, 225]
ripeness, τ = 0.9	1.32, 116 [1.33, 119]	1.32, 168 [1.32, 178]	1.22, 256 [1.22, 261]
ripeness, τ = 0.95	1.34, 112 [1.33, 119]	1.32, 165 [1.31, 174]	1.17, 322 [1.17, 328]
static share, τ = 0.75	1.46, 102 [1.47, 111]	1.48, 147 [1.50, 148]	1.47, 189 [1.47, 203]
static share, τ = 0.85	1.30, 132 [1.29, 143]	1.32, 179 [1.32, 184]	1.29, 230 [1.30, 243]
static share, τ = 0.9	1.26, 139 [1.26, 141]	1.25, 211 [1.26, 209]	1.22, 272 [1.22, 279]

packed	tiger	coat of arms	world map
static share, $\tau = 0.95$	1.25, 142 [1.25, 142]	1.26, 207 [1.26, 210]	1.16, 359 [1.16, 366]
Bayesian, $\tau = 0.75$	1.48, 102 [1.46, 106]	1.50, 142 [1.50, 148]	1.47, 186 [1.47, 197]
Bayesian, $\tau = 0.85$	1.33, 116 [1.31, 122]	1.32, 171 [1.31, 180]	1.29, 225 [1.29, 231]
Bayesian, $\tau = 0.9$	1.32, 117 [1.30, 121]	1.29, 184 [1.29, 191]	1.22, 263 [1.22, 274]
Bayesian, $\tau = 0.95$	1.34, 117 [1.31, 122]	1.28, 194 [1.28, 198]	1.16, 360 [1.16, 370]

small	tiger	coat of arms	world map
ripeness, $\tau = 0.75$	1.32, 102 [1.31, 110]	1.39, 148 [1.40, 153]	1.39, 213 [1.39, 218]
ripeness, $\tau = 0.85$	1.21, 117 [1.22, 116]	1.25, 172 [1.27, 179]	1.22, 253 [1.23, 256]
ripeness, $\tau = 0.9$	1.21, 113 [1.23, 118]	1.25, 175 [1.26, 180]	1.16, 297 [1.16, 300]
ripeness, $\tau = 0.95$	1.21, 112 [1.22, 117]	1.26, 176 [1.26, 179]	1.13, 338 [1.13, 334]
static share, $\tau = 0.75$	1.29, 107 [1.30, 115]	1.41, 152 [1.41, 163]	1.40, 224 [1.40, 233]
static share, $\tau = 0.85$	1.17, 129 [1.17, 133]	1.23, 187 [1.24, 202]	1.23, 264 [1.23, 271]
static share, $\tau = 0.9$	1.14, 146 [1.14, 147]	1.17, 223 [1.17, 231]	1.16, 313 [1.16, 315]
static share, $\tau = 0.95$	1.13, 148 [1.15, 145]	1.17, 228 [1.17, 231]	1.10, 412 [1.11, 412]
Bayesian, $\tau = 0.75$	1.33, 104 [1.34, 109]	1.40, 154 [1.41, 160]	1.39, 225 [1.40, 234]
Bayesian, $\tau = 0.85$	1.26, 111 [1.28, 120]	1.26, 179 [1.26, 182]	1.23, 267 [1.23, 273]
Bayesian, $\tau = 0.9$	1.26, 115 [1.28, 118]	1.26, 179 [1.26, 182]	1.16, 308 [1.16, 311]
Bayesian, $\tau = 0.95$	1.23, 111 [1.24, 118]	1.26, 178 [1.27, 183]	1.12, 369 [1.12, 369]

text	64 KiB	512 KiB
ripeness, $\tau = 0.75$	1.41, 36.4 [1.45, 37.9]	1.16, 49.4 [1.21, 49.5]
ripeness, $\tau = 0.85$	1.29, 40.0 [1.26, 40.2]	1.16, 49.5 [1.19, 49.9]
ripeness, $\tau = 0.9$	1.20, 42.6 [1.19, 42.8]	1.16, 49.5 [1.17, 50.4]
ripeness, $\tau = 0.95$	1.18, 43.5 [1.17, 43.7]	1.13, 53.0 [1.13, 53.5]
static share, $\tau = 0.75$	1.43, 38.7 [1.44, 40.5]	1.18, 51.7 [1.20, 51.8]
static share, $\tau = 0.85$	1.28, 41.7 [1.26, 42.3]	1.17, 52.7 [1.19, 52.6]
static share, $\tau = 0.9$	1.20, 44.4 [1.19, 44.0]	1.16, 54.0 [1.17, 54.2]
static share, $\tau = 0.95$	1.15, 49.1 [1.15, 48.7]	1.12, 56.9 [1.12, 56.3]
Bayesian, $\tau = 0.75$	1.46, 37.2 [1.45, 39.3]	1.19, 50.7 [1.22, 51.0]
Bayesian, $\tau = 0.85$	1.28, 41.7 [1.27, 41.2]	1.19, 50.4 [1.21, 51.1]
Bayesian, $\tau = 0.9$	1.20, 44.3 [1.19, 44.3]	1.18, 51.9 [1.18, 51.8]
Bayesian, $\tau = 0.95$	1.19, 44.6 [1.19, 44.0]	1.12, 55.3 [1.12, 55.0]

Chapter 34

The file as one sequence, measured

34.1 Executive summary

On the drawings and texts of Cleaning edited drawings and texts, under the same edits, the one-sequence branch keeps as few live pages as main's best layout at equal writes, or fewer, and its churn floor reaches further. kladde-rs's branch `one-sequence` implements the design of one sequence, which was a proposal when this page was written, in all four of its stages, and ran svg-bench and text-bench at the six settings of the churn floor that main ran, with three bounds L of its span policy. With its default bound, $L = 64$ bytes, it keeps 1 to 10 % fewer live pages than main's small layout at equal writes on the tiger, 1 to 2 % fewer on the world map, and as many, within 1 %, on the coat of arms; 2 to 17 % fewer than the packed layout, but for the world map at $\lambda = 1$, and 35 to 61 % fewer than the base layout. On the texts, it keeps 2 to 3 % fewer than main where their curves overlap on the 64 KiB text, and as many on the 512 KiB text, where main's controller had cut its budget to a page. At $\lambda = 0.1$, it keeps 1.06 to 1.11 times the drawings' live size and 1.07 and 1.05 times the texts', where main reached 1.08 to 1.09 in the small layout, 1.13 to 1.19 in the packed one, and 1.13 and 1.09 on the texts.

The sequence is as compact as main's small layout, and half its base layout. The drawings' live size, the sequence's bytes with the schema chain's, is 46, 128 and 942 KiB, within 0.4 % of the small layout's live allocation bytes, 1.3 to 2.4 % below the packed layout's, and 0.50 to 0.70 times the base layout's. So the ratios of live pages to live size compare one sequence with main's small layout byte for byte, and its live pages compare with main's other layouts on the same drawing without a ratio.

The journal writes 14 to 27 % less, and splices restate nothing behind them. Relative positions shorten a drawing edit's journal bytes from 37 to 51 per edit in main's small layout to 31 to 43, and a keystroke's from 13.2 to 9.6, as the journal's arithmetic expected. On the texts, the leaves take 6 to 18 bytes of each keystroke's writes where main's took 15 to 30, since a splice states one to three entries rather than every statement behind it.

The span policy's bound decides much of the writes: a merge costs the bytes it writes again, and the garbage it leaves for the cleaner. At $L = 256$, the proposal's sizing and the branch's first default, the cleaner moves 1.6 to 2.1 times the bytes it moves at $L = 64$ on the drawings and texts at $\lambda = 0.25$; at $L = 64$, every setting of the churn floor writes 4 to 33 % less, for about as many live pages at equal writes, which made 64 the default. At $L = 64$, the leaves take 8 to 26 % of the drawings' live pages and 4 to 10 % of the texts', against 2 to 8 % and 2 to 4 % at $L = 256$, since short spans keep more entries and more payloads inline; $L = 32$ traces nearly the same curves as 64.

Edits stay fast, flushes short, and opening fast. At the default bound, kladde's calls take 39,000 to 123,000 edits per second on the drawings, as many as in main's small layout or up to 1.6 times as many, with median flushes of 2.2 to 5.0 ms, against 2.6 to 10.8; on the texts, 691,000 to 837,000 keystrokes per second, 14 to 30 % fewer than main, with flushes of 1.9 to 2.8 ms, against 6.8 to 9.2. Reopening an edited file takes 0.8 to 7.7 ms, as on main.

Where it falls short of the proposal: the span table holds two levels only, a span costs about 90 bytes of memory rather than 32, and the cleaner, the evacuation and the walk are main's, without the refinements the proposal names; kladde-rs's implementation notes record every deviation.

34.2 What was measured

kladde-rs's branch `one-sequence`, at `b2d0649`, ran the benchmarks of Cleaning edited drawings and texts unchanged but for what one sequence removes. The branch starts from `defrag-shifts` at `b1a7f90`, as the proposal asked, and takes eight commits: seven that implement the store, the schema, persist, the derive and the facade, the containers, `text-bench`, and kladde-svg, and one, `b2d0649`, that sets the span policy's defaults to what this page measured. kladde-svg has one layout, since every list is a `PersistableVec` and every string a `PersistableString`, inline in its parent; `text-bench` splices the store's sequence itself, as it spliced one allocation on main.

The edits are the same, edit for edit. svg-bench ran each drawing for main's number of edits, 31,613, 41,130 and 177,831, with the same seed, the same mix and the same focus, and every run ended with main's canonical SVG, of 59,005, 184,541 and 1,069,891 bytes; text-bench typed main's 100,000 keystrokes, and the texts ended at main's 241,322 and 700,340 bytes. The store flushed whenever the journal outgrew four pages, `--journal-budget 4`, and at every row, every 1000 edits or keystrokes.

Each workload ran at six settings of the churn floor, with three bounds of the span policy: 90 runs, eight at a time.

span policy	`L`, `Options::span_min`	`Options::rewrite_below`	churn floor λ
`L` = 256	256	128	1, 0.5, 0.35, 0.25, 0.15, 0.1
`L` = 64, the default	64	64	as above
`L` = 32	32	32	as above

The bound is the span policy's: two neighbouring spans whose entries leave the header for a leaf are merged if they are shorter than `L` together. `rewrite_below` is the threshold for overwrites: where an edit cuts into a `Ref` span shorter than it, the flush writes the span's bytes again, with the edit's, rather than split it. It applies to the span being cut, not to the parts a cut leaves, so shorter `Ref`s still arise, from splits of longer spans and from evacuation, as do short inline spans of new bytes; the bound merges those as their entries leave the header, 0.5 to 9.3 bytes per edit at `L` = 64 in the span policy's table, so the two constants matter even where they are equal. The sweep ties them to keep its grid small, after tuning runs at λ = 1, which `data/` does not keep, had varied them apart: over four workloads, `rewrite_below` moved the bytes written per edit by at most 7 %, and `L` by up to 21 %. The span policy's other constants kept their defaults: inline spans up to 64 bytes, 16 flushes to count as cold for the rotating walk, which looks at 256 spans per flush, and μ = 0.02. The cleaner is main's churn floor, with its budget controller aiming the file at a fill of 0.8, its free filling, its compaction mode and truncation, over pages of spans.

main is the churn floor's runs of Cleaning edited drawings and texts, on `defrag-shifts` at `7eb2ec4`, in the drawings' base, packed and small layouts and on the texts, from the tables in `data/cleaning-edits/`.

The measures are main's, with the live size taken from the sequence. The *live size* is the sequence's bytes and the schema chain's, the column `alloc_bytes` of the tables; *live pages* are the data pages and the leaves, without the header and the journal, and the branch writes no interior pages of the span table. Space is averaged over the second half of each run, and writes are what the store wrote during that half, per edit or keystroke: the journal's bytes and every page written, headers included, at 4,096 bytes each. The tables keep kladde-bench's columns, so that `tools/plot-evaluation.py` reads them: `statements` and `fragments` count the spans, `allocations` is 0, `window_restated` counts every entry stated, `defrag_rewrites` and `defrag_bytes` the span policy's merges and the bytes they wrote again, and `restated_bytes` the bytes of short spans written again where an edit cut into them; columns of the span policy's own follow.

The runs were built with `--release` by Rust 1.97.1 and ran on the Intel Core i7-1165G7 of Cleaning edited drawings and texts, four cores and eight threads, 16 GB of memory, under Linux 7.0 in a development container. The benchmarks are deterministic but for their times; the times come from two further passes, one run at a time, of the default bound at λ = 1, 0.25 and 0.1. `data/one-sequence/` keeps every run's table, in a directory named `<policy>-lambda<λ>-<layout>`, with `seq256`, `seq64` and `seq32` for the bounds and `one` and `text` for the layouts, `logs/` beside them the end of each run's output, and `timing/`

and `timing2/` the two timing passes' tables and output, from which the rates, the flush times and the reopen times come.

To reproduce a run, build the benchmarks on the branch, and run them as main's were, with the bound's options:

```
cargo build --release -p kladde-svg --bin svg-bench -p kladde-bench --bin text-bench
target/release/svg-bench crates/kladde-svg/corpus/drawings/world-map.svg out/ \
    --edits 177831 --journal-budget 4 --churn-floor 0.25 --span-min 64 --rewrite-below 64
target/release/text-bench out/ --size 524288 --journal-budget 4 --churn-floor 0.25 \
    --span-min 64 --rewrite-below 64
```

The figures come from the tables of both pages, each run labelled `<policy>/lambda/<λ>/<layout>`:

```
runs=()
for dir in content/evaluation/data/cleaning-edits/main-*-*/ content/evaluation/data/one-sequence/seq*-
    IFS=- read -r p k l <<< "$(basename $dir)"; knob=${k%%[0-9]*}
    runs+=("$p/$knob/${k#$knob}/$l=$dir")
done
tools/plot-evaluation.py --only seq-svg-tradeoff,seq-text-tradeoff \
    --out content/evaluation/figures/one-sequence "${runs[@]}"
```

34.3 The trade-off between space and writes

With the default bound, one sequence's curve lies on or below main's lowest on every drawing, and on or below main's on the texts, and reaches further down. Each curve runs through the six settings of the churn floor, from $\lambda = 1$, hollow, to $\lambda = 0.1$; main's are dashed, one per layout. Each drawing takes a column: above, live pages over the live size; below, the live pages in KiB, which compare the layouts on the same drawing, whatever their encodings:

- **Against main's small layout, whose live size one sequence's matches, the default bound keeps 1 to 10 % fewer live pages on the tiger, 1 to 2 % fewer on the world map, and on the coat of arms between 0.7 % fewer and 0.4 % more, at equal writes.** On the tiger the gap closes as the churn floor falls: 10 % at $\lambda = 0.5$, 1 % at $\lambda = 0.1$.
- **Against the packed layout, it keeps 9 to 17 % fewer live pages on the tiger, 3 to 7 % fewer on the coat of arms, and 2 to 4 % fewer on the world map at equal writes**, but at $\lambda = 1$ on the world map, where the packed layout, whose writes start lowest, keeps 3 % fewer.
- **Against the base layout, it keeps 35 to 61 % fewer live pages in KiB**, since the base layout's live size is 1.4 to 2 times larger, and 7 to 22 % fewer relative to each layout's own live size. On the tiger, the base layout writes more at its least than one sequence does at its most, so the curves do not overlap.
- **At its lowest writes, $\lambda = 1$, an edit writes 84, 135 and 225 bytes, against 96, 142 and 215 in the small layout and 96, 121 and 164 in the packed one**, and keeps 1.39, 1.39 and 1.28 times the live size, against 1.39, 1.43 and 1.34 and 1.52, 1.56 and 1.40.
- **The bound matters less than the churn floor**: $\mathsf{L} = 32$ and 64 trace nearly one curve; $\mathsf{L} = 256$ writes more for the same live pages on the coat of arms and the world map, while on the tiger its curve crosses $\mathsf{L} = 64$'s; at the lowest λ, it reaches furthest, 1.05 to 1.09 times the live size.

The equal-write comparisons interpolate main's curve between its settings, at every setting of one sequence whose writes lie within it.

On the texts, a panel per text gives the live pages over the live size:

- **On the 64 KiB text, the default bound keeps 2.0 to 2.6 % fewer live pages than main where the curves overlap, between 38 and 48 bytes per keystroke, and writes from 30 bytes per keystroke, where main's least is 37.** Below main's curve, it goes on to 1.07 times the live size at 74 bytes per keystroke, where main's $\lambda = 0.1$ kept 1.13 at 48.
- **On the 512 KiB text, main's six settings lie within 2 bytes per keystroke of each other, at 1.09 to 1.10 times the live size, and one sequence's curve passes through them**: between $\lambda = 0.5$ and 0.35 it keeps 1.10 and 1.09 times the live size at 47 and 53 bytes per keystroke. main's controller had cut its budget to one page, since main's file passed the target fill of 0.8 after about 11,000 keystrokes;

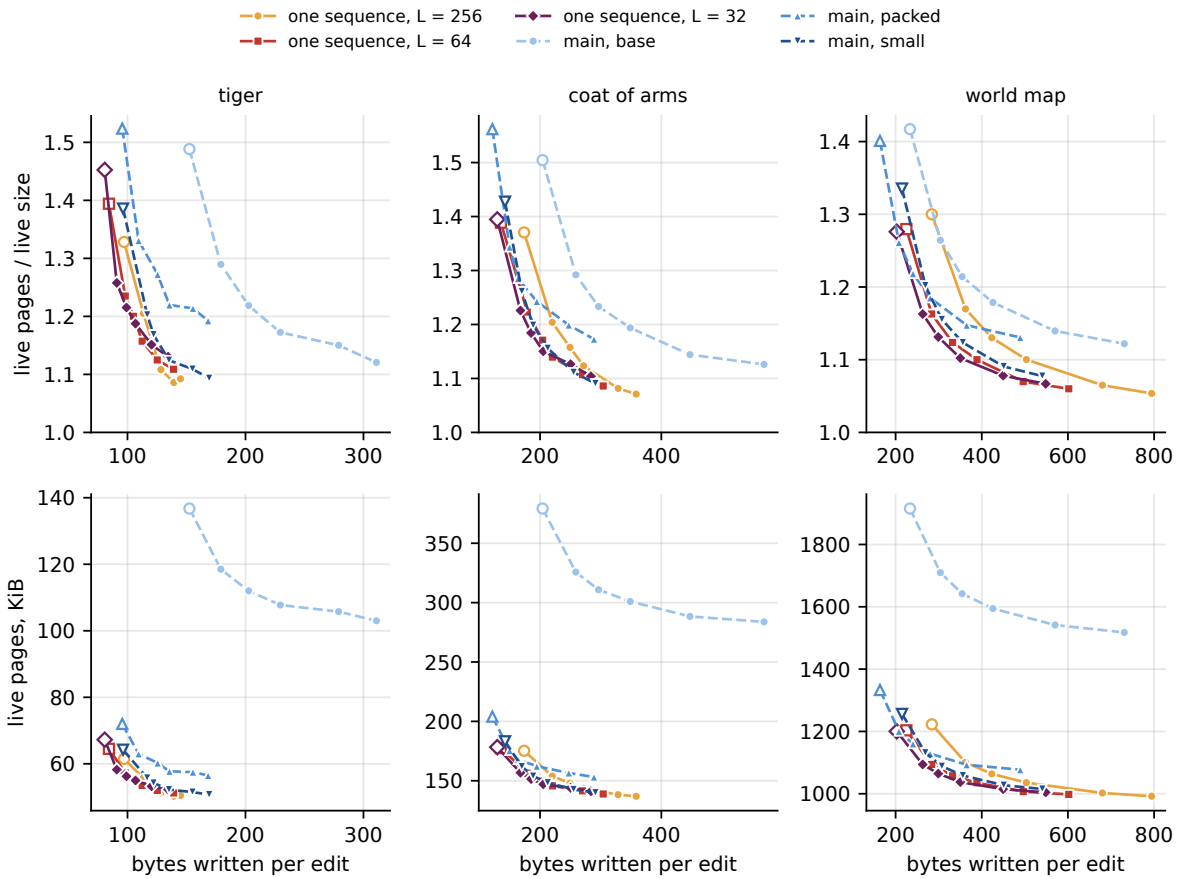


Figure 34.1: Live pages over live size, and live pages in KiB, against the bytes written per edit, a column per drawing, for one sequence at L = 256, 64 and 32 and main's churn floor in its base, packed and small layouts, each at λ = 1, 0.5, 0.35, 0.25, 0.15 and 0.1

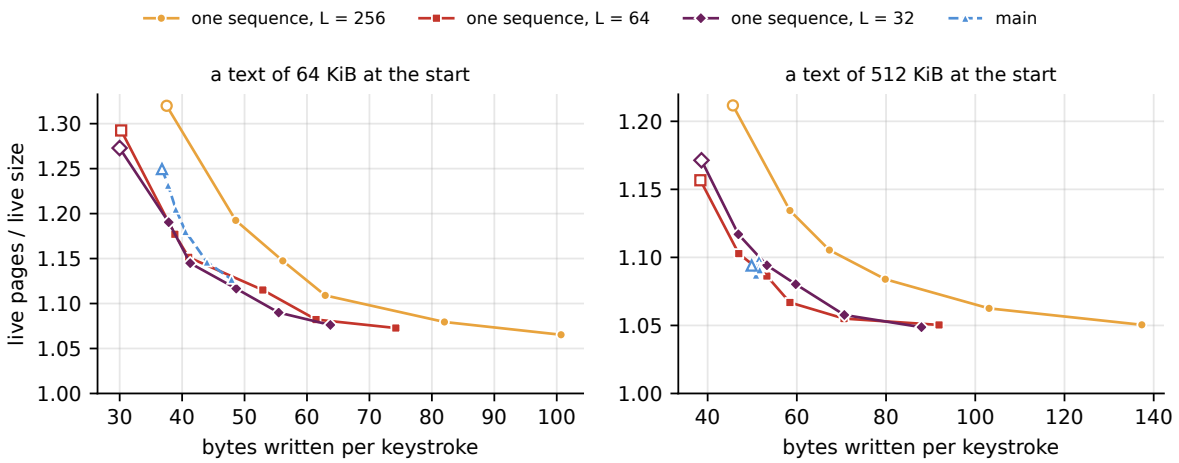


Figure 34.2: Live pages over live size against the bytes written per keystroke, a panel per text, for one sequence at L = 256, 64 and 32 and main's churn floor, each at λ = 1, 0.5, 0.35, 0.25, 0.15 and 0.1

one sequence's file stays below it, its budget at 223 to 256 pages, so every λ cleans as its own setting says.

34.4 The live size: the encoding

One sequence's live size is the small layout's, within 0.4 %, and 1.3 to 2.4 % below the packed layout's: what the small layout kept inline is everything here, and what it described with statements and pointers costs nothing. The live size is averaged over the second half of the runs; it does not depend on the bound or on λ:

workload	edited SVG or text at the end, KiB	main, base	main, packed	main, small	one sequence
tiger	58	92	47	46	46.3
coat of arms	180	252	131	128	127.9
world map	1,045	1,353	954	943	941.6
text, 64 KiB	236	193			192.8
text, 512 KiB	684	642			642.0

On main, the live size counts the bytes of the allocations, and the statements that describe them come on top, as live bytes in leaves; here the entries that describe the spans come on top in the same way. Live pages gives both.

34.5 Live pages

At λ = 0.25, one sequence keeps 13.4, 36.4 and 258.8 live pages on the drawings, against 13.1, 37.1 and 264.6 in main's small layout, with more of them leaves: 3.4, 7.1 and 20.9, against 0.5, 2.0 and 10.6. Each cell gives the live pages, and in parentheses the leaves among them, at the default bound; the first column gives the live bytes they hold, in data pages and in leaves, which hardly change with λ:

workload	live bytes, KiB	λ = 1	λ = 0.25	λ = 0.1
tiger	35 + 11	16.1 (3.6)	13.4 (3.4)	12.8 (3.4)
coat of arms	107 + 24	44.4 (8.0)	36.4 (7.1)	34.7 (7.0)
world map	887 + 74	300.9 (25.8)	258.8 (20.9)	249.5 (21.2)
text, 64 KiB	177 + 19	62.2 (6.8)	53.7 (5.6)	51.7 (5.6)
text, 512 KiB	625 + 25	185.7 (8.7)	171.3 (7.1)	168.6 (7.5)

Over the live size, the same live pages:

workload	live size, KiB	λ = 1	λ = 0.25	λ = 0.1
tiger	46	1.39	1.16	1.11
coat of arms	128	1.39	1.14	1.09
world map	942	1.28	1.10	1.06
text, 64 KiB	193	1.29	1.12	1.07
text, 512 KiB	642	1.16	1.07	1.05

The leaves hold more than in main's small layout because they hold content. A new run of bytes up to 64 long is an inline span, its bytes in its entry, and a span shorter than the bound may stay; so the leaves carry 11, 24 and 74 KiB of the drawings, where the small layout's statements took 1, 7 and 36, and the data pages correspondingly less: 35, 107 and 887 KiB, against 45, 127 and 937. The live bytes differ from

the live size by −0.8 to +3.0 %: the entries' framing comes on top, and the bytes inline in the header's entries, up to a page of them, are left out, as the header is left out of the live pages; on the tiger, the second outweighs the first.

The pages fill as main's do under the churn floor: at λ = 0.25, the data pages 89 to 96 % full, the leaves 80 to 89 %. Each cell gives the fill of the data pages, of the leaves, and of all live pages, in percent, at the default bound:

workload	λ = 1	λ = 0.25	λ = 0.1
tiger	71 / 73 / 72	89 / 80 / 86	93 / 83 / 90
coat of arms	74 / 75 / 74	92 / 86 / 91	96 / 90 / 95
world map	81 / 70 / 80	93 / 89 / 93	97 / 94 / 97
text, 64 KiB	80 / 71 / 79	92 / 85 / 92	96 / 87 / 95
text, 512 KiB	89 / 71 / 88	96 / 86 / 95	97 / 90 / 97

The proposal's worked example put a large file's live pages at about 1.12 times its sequence with pages 90 % live; the world map, at 93 % live, comes to 1.10, and the 512 KiB text, at 95 %, to 1.07. The tiger and the coat of arms, with pages 86 and 91 % live, come to 1.16 and 1.14, since their leaves, a quarter and a fifth of their live pages, are only 80 and 86 % full.

34.6 The span policy

Merging spans costs writes twice: the bytes a merge writes again, and the old bytes it leaves dead in their data pages, which the cleaner then moves survivors to reclaim. At λ = 0.25, each row gives the spans the chains reach and their mean length, the leaves' share of the live pages, the spans split per edit, and per edit the bytes the flush wrote again: short spans an edit cut into, spans merged as their entries left the header, spans the rotating walk merged, and survivors the cleaner moved; and the bytes written per edit in all:

workload	L	spans	mean span, bytes	leaves, % of live pages	splits	rewritten	merged at eviction	merged by the walk	evacuated	written
tiger	256	515	92	7.8	0.23	8.4	8.8	0.4	49.9	128
tiger	64	671	71	25.7	0.24	5.6	2.5	1.3	30.7	112
tiger	32	906	52	25.0	0.27	3.5	1.2	1.6	26.7	107
coat of arms	256	861	152	5.6	0.43	12.4	23.7	6.8	139.6	272
coat of arms	64	1,294	101	19.5	0.45	8.4	4.1	11.8	81.5	220
coat of arms	32	1,703	77	17.0	0.50	4.8	1.9	11.7	75.6	205
world map	256	2,257	427	2.3	0.73	30.0	47.3	36.7	258.0	503
world map	64	3,660	263	8.1	0.74	19.2	9.3	36.6	136.4	389
world map	32	5,121	188	8.4	0.79	11.3	4.7	35.2	114.5	350
text, 64 KiB	256	829	238	3.7	0.04	0.7	5.5	5.1	27.9	63
text, 64 KiB	64	1,211	163	10.3	0.04	0.3	0.7	6.2	17.6	53

workload	L	spans	mean span, bytes	leaves, % of live pages	splits	rewritten	merged at eviction	merged by the walk	evacuated	written
text, 64 KiB	32	1,428	138	9.6	0.04	0.2	0.2	6.1	15.6	49
text, 512 KiB	256	1,493	440	2.3	0.04	0.3	5.6	11.5	35.9	80
text, 512 KiB	64	1,844	357	4.1	0.04	0.1	0.5	12.7	16.9	58
text, 512 KiB	32	2,099	313	4.1	0.04	0.1	0.1	12.1	19.3	60

- **From L = 256 to 64, the merges at eviction write 3.5 to 11 times less, and the cleaner moves 1.6 to 2.1 times less**, since every merge leaves the bytes it copied dead where they lay; the rewrites of short spans fall with `rewrite_below`, and the walk, which merges by temperature and μ rather than by the bound, writes about the same. So every setting of the churn floor writes 4 to 33 % less at L = 64, the least on the tiger at λ = 0.1 and the most on the 512 KiB text at λ = 0.1.
- **The spans stay short**: 71 to 263 bytes on average on the drawings at L = 64, and 163 and 357 on the texts, against the proposal's 128 bytes where the sequence is hot and a page where it is cold. The rotating walk merges cold runs, 1.3 to 36.6 bytes per edit on the drawings, whatever the bound, and still the mean span stays below a page by far; this page has not traced which spans stay short, or why the walk leaves them.
- **Short spans move content into the leaves, and add little to the live pages**: at L = 64, the leaves take 8 to 26 % of the drawings' live pages, where they took 2 to 8 % at L = 256, and the live pages at equal writes stay within a few percent. A bound of 64 lets spans of up to 64 bytes stay inline, in their entries, which L = 256 merged into data pages; the leaves then hold those spans' content as well as the description.
- **A splice states one to three entries, and a flush folds the keystrokes at one place into one span**: on the texts, a flush states 0.5 entries per keystroke at λ = 1 and splits 0.04 spans per keystroke, whatever the bound.

L = 32 writes 3 to 10 % less than 64 on the drawings, and between 14 % less and 2 % more on the texts, for about the same live pages at equal writes, with 1.1 to 1.4 times the spans; the default stays at 64, since the bound's guarantee, a mean span in the leaves of at least `L / 2`, weakens as L falls, and nothing measured here asks for it.

34.7 Files

Free pages make up as much of the files as on main: during the edits, the tiger's file holds 25 to 34 free pages beside 13 to 16 live ones, the coat of arms' 31 to 53 beside 35 to 44, and the world map's 75 to 106 beside 250 to 301. Each cell gives the file's pages during the edits, averaged over the second half, and in parentheses the free ones among them, at the default bound:

workload	λ = 1	λ = 0.25	λ = 0.1
tiger	44 (25)	47 (31)	50 (34)
coat of arms	78 (31)	82 (43)	90 (53)
world map	395 (91)	336 (75)	358 (106)
text, 64 KiB	82 (16)	86 (30)	92 (37)
text, 512 KiB	218 (29)	215 (41)	231 (60)

main's small layout held 45, 44 and 49 pages on the tiger, 78, 79 and 89 on the coat of arms, and 408, 346

and 355 on the world map, at $\lambda = 1$, 0.25 and 0.1: the same files, within a few pages, since the cleaner, its truncation and its compaction mode are main's.

Closed, the files take what main's take, after the last edit and `close`, in KiB:

workload	edited SVG or text	$\lambda = 1$	$\lambda = 0.25$	$\lambda = 0.1$	main, small, at $\lambda = 1$, 0.25, 0.1
tiger	58	152	156	176	136, 164, 168
coat of arms	180	288	260	264	292, 276, 324
world map	1,045	1,208	1,176	1,300	1,280, 1,120, 1,196
text, 64 KiB	236	376	416	400	384, 360, 400
text, 512 KiB	684	916	876	988	848, 868, 876

As on main, the closed sizes depend on where the last live page happens to lie when the edits end, and vary by up to 16 % between neighbouring settings.

34.8 Writes

The journal and the headers write 13 to 21 % less than in main's small layout, and the leaves more on the drawings and less on the texts. Each cell gives the bytes per edit, or per keystroke on the texts, over the second half of the run, at the default bound; the first columns give the journal and the headers, which do not depend on λ, and main's beside them:

workload	journal + headers	main, small	$\lambda = 1$	$\lambda = 0.25$	$\lambda = 0.1$
tiger	31.3 + 9.5	36.6 + 11.3	84	112	139
coat of arms	37.0 + 12.1	44.7 + 12.7	135	220	304
world map	42.9 + 13.7	51.0 + 14.4	225	389	602
text, 64 KiB	9.6 + 4.1	13.2 + 4.1	30	53	74
text, 512 KiB	9.5 + 4.1	13.2 + 4.1	38	58	92

Each cell splits the rest into data pages and leaves, with main's small layout in the same cells of the cleaning page beside it:

workload	$\lambda = 1$	$\lambda = 0.25$	$\lambda = 0.1$	main, small, at $\lambda = 1$, 0.25, 0.1
tiger	26 + 18	46 + 25	68 + 30	41 + 7, 79 + 8, 114 + 8
coat of arms	56 + 31	119 + 52	182 + 73	63 + 21, 129 + 27, 208 + 26
world map	106 + 62	214 + 119	363 + 183	95 + 54, 196 + 95, 336 + 139
text, 64 KiB	11 + 6	28 + 12	45 + 16	4.6 + 14.9, 8.0 + 15.2, 15.3 + 15.3
text, 512 KiB	19 + 6	33 + 12	60 + 18	4.8 + 27.8, 4.7 + 29.5, 4.8 + 29.5

- **A drawing's edit appends 31 to 43 bytes to the journal, against 37 to 51**, and a keystroke 9.6, against 13.2: a record names its position relative to where the record before left off, with its kind in the same varint, where main's records named an allocation and an offset in it.
- **On the drawings, the leaves take 29 to 41 % of what goes to data pages and leaves, against 7 to 36 % in the small layout**: a flush states 1.2 to 3.4 entries per edit at $\lambda = 1$ and 1.5 to 8.9 at $\lambda = 0.1$, of which the cleaner's restatements of survivors are the growing part, and the leaves hold inline content that the small layout kept in data pages.
- **On the texts, the leaves take 6 to 18 bytes per keystroke, against main's 15 to 30, which restated the statements behind every splice**; the data pages take more, since the cleaner, whose budget main's controller had cut on the 512 KiB text, cleans at every λ here.

Every setting writes far less than saving the file would. Saving the canonical SVG every 100 edits would write 590, 1,845 and 10,699 bytes per edit, and saving the text every 1000 keystrokes 241 and 700 bytes per keystroke; at $\lambda = 0.25$, one sequence writes 112, 220 and 389 bytes per edit, and 53 and 58 per keystroke.

34.9 Times

Edits are as fast as in main's small layout or faster, flushes mostly shorter, and opening takes as long; keystrokes are 14 to 30 % slower than on main, and their flushes a fifth to two fifths as long. Each cell gives the edits per second in kladde's calls, in thousands, or on the texts the keystrokes per second, counting the flushes that a full journal starts within an edit; the median time of the flushes at the rows, in milliseconds; and the median time to reopen the closed file, in milliseconds; each the mean of the two timing passes, at the default bound, and main's small layout and main on the texts, from the cleaning page:

workload	$\lambda = 1$	$\lambda = 0.25$	$\lambda = 0.1$	main, small or on the texts, at $\lambda = 1$, 0.25, 0.1
tiger	115 / 3.0 / 0.96	123 / 3.0 / 0.91	115 / 3.1 / 0.89	111 / 2.6 / 0.8, 101 / 3.6 / 0.8, 99 / 3.9 / 0.8
coat of arms	84 / 2.2 / 2.3	81 / 2.7 / 1.8	75 / 3.4 / 1.9	74 / 4.2 / 1.8, 67 / 5.0 / 1.7, 64 / 5.1 / 1.7
world map	48 / 3.6 / 7.3	44 / 4.2 / 7.3	39 / 5.0 / 7.7	37 / 6.1 / 8.0, 29 / 7.8 / 7.9, 24 / 10.8 / 7.6
text, 64 KiB	735 / 2.4 / 0.84	782 / 2.6 / 0.75	837 / 2.8 / 0.76	982 / 6.9 / 0.7, 1,016 / 6.8 / 0.7, 968 / 7.2 / 0.7
text, 512 KiB	804 / 1.9 / 1.8	691 / 2.5 / 1.7	770 / 2.5 / 1.5	958 / 9.2 / 1.4, 993 / 8.9 / 1.4, 966 / 9.0 / 1.5

- **On the world map, edits run 1.3 to 1.6 times as fast as in the small layout, and flushes take 0.46 to 0.59 times as long**, the gap growing as the churn floor falls; on the tiger and the coat of arms, edits are 4 to 22 % faster, and flushes 17 to 48 % shorter, but at $\lambda = 1$ on the tiger, where they take 15 % longer. This page has not timed the parts of an edit or a flush apart.
- **On the texts, a keystroke takes 1.2 to 1.4 µs in the store, against 1.0 on main**, and the flushes at the rows take 1.9 to 2.8 ms, against 6.8 to 9.2, since a thousand keystrokes' splices state a few spans rather than every statement behind them. This page has not traced where a keystroke's extra time goes.
- **Reopening takes 0.9 ms on the tiger, 1.8 to 2.3 on the coat of arms, 7.3 to 7.7 on the world map, and 0.8 and 1.5 to 1.8 on the texts**, as in main's small layout within half a millisecond: it reads the span table, walks the chain, and decodes the sequence into the drawing.

The benchmarks' processes peak at 3.9 to 6.1 MB of resident memory on every workload but the world map, which takes 15.7 MB, at $\lambda = 0.25$, sampled from `/proc` every 10 ms. Of those peaks, 2.6 to 3.0 MB are the binary and its libraries, mapped from disk, and 1.3 to 3.0 MB anonymous memory, which holds at 1.2 to 2.7 MB over the second half of the runs. The world map's anonymous memory grows from 7.5 MB once the file is created to 10.4 MB, where it holds over the second half; valgrind's massif finds 8.3 MB of heap there, at 90,000 edits, a fifth of it the store's, and the allocator's overhead and the freed memory it keeps make the rest:

owner	MB	what
the drawing in memory	4.7	the children's vectors, 1.9 MB, since a `Node` holds an `Element` inline and takes 328 bytes; the path segments, 1.7 MB; the elements' other vectors; and every vector's spare capacity

owner	MB	what
the store	1.7	the journal's byte arena, 0.79 MB, still at the capacity of the splice that created the file; the span table, 0.39 MB, and the rope, 0.33 MB, both doubled to 8,192 slots for 3,660 spans; resident table pages and the flush's page cache, 0.1 to 0.2 MB
svg-bench	1.5	the SVG's text, 1.1 MB, kept for its summary, and its index of elements and paths, 0.4 MB
below massif's threshold	0.4	

The peak comes in svg-bench's epilogue, which writes the edited drawing's canonical SVG into a string of 2.1 MB and then reopens the file, decoding a second drawing: 10.6 MB of heap, which with the allocator's overhead and the mapped binary makes the 15.7 MB.

Most of it could go, and little of it is the store's. A `Box<Element>` in kladde-svg's `Node` would shrink each node from 328 bytes to 48, and vectors built at their final length would lose their spare capacity. kladde-types' vector spends 40 bytes on its index of sizes even where it is empty or its elements are fixed-size, 0.86 MB over the world map's 13,385 vectors: the tree's cached total and the count's length need no fields, which leaves the 24 bytes of an empty `Vec`, or 8 with a thin one; and a Fenwick tree over blocks of 32 elements, scanning up to 31 elements' sizes within a block, would cost an eighth of a byte per element rather than four. The store's journal arena should shrink to the journal's budget after a flush, and its span table and rope could grow in smaller steps, or become a B-tree, which would bring the store's share to about half a megabyte for this drawing; and svg-bench could drop its copy of the SVG after parsing. main's memory was not measured, so this page cannot say how one sequence compares with it.

Identical runs differed by more than on the cleaning page. The two timing passes differed in the edit rate by 8 % at the median and 32 % at most, in the median flush by 5 % and 26 %, and in the reopen times by 16 % and 33 %, the largest on the 512 KiB text's rates and on the coat of arms' reopens, which take about two milliseconds. The reopen times are medians of 21 opens each.

34.10 The sweep in numbers

Every setting's steady state: the live pages over the live size, and the bytes written per edit or keystroke. These are the points of the trade-off figures; main's are in the cleaning page's sweep.

drawings	tiger	coat of arms	world map
$L = 64, \lambda = 1$	1.39, 84	1.39, 135	1.28, 225
$L = 64, \lambda = 0.5$	1.24, 98	1.22, 179	1.16, 284
$L = 64, \lambda = 0.35$	1.20, 105	1.17, 204	1.12, 332
$L = 64, \lambda = 0.25$	1.16, 112	1.14, 220	1.10, 389
$L = 64, \lambda = 0.15$	1.12, 125	1.11, 270	1.07, 496
$L = 64, \lambda = 0.1$	1.11, 139	1.09, 304	1.06, 602
$L = 32, \lambda = 1$	1.45, 81	1.39, 129	1.28, 203
$L = 32, \lambda = 0.5$	1.26, 91	1.23, 168	1.16, 263
$L = 32, \lambda = 0.35$	1.22, 99	1.18, 185	1.13, 299
$L = 32, \lambda = 0.25$	1.19, 107	1.15, 205	1.10, 350
$L = 32, \lambda = 0.15$	1.15, 120	1.13, 250	1.08, 450
$L = 32, \lambda = 0.1$	1.13, 134	1.10, 284	1.07, 548
$L = 256, \lambda = 1$	1.33, 97	1.37, 174	1.30, 284
$L = 256, \lambda = 0.5$	1.21, 113	1.20, 220	1.17, 362
$L = 256, \lambda = 0.35$	1.16, 121	1.16, 249	1.13, 423
$L = 256, \lambda = 0.25$	1.11, 128	1.12, 272	1.10, 503
$L = 256, \lambda = 0.15$	1.09, 139	1.08, 329	1.06, 680

drawings	tiger	coat of arms	world map
$\mathsf{L} = 256, \lambda = 0.1$	1.09, 145	1.07, 359	1.05, 794

texts	64 KiB	512 KiB
$\mathsf{L} = 64, \lambda = 1$	1.29, 30.2	1.16, 38.4
$\mathsf{L} = 64, \lambda = 0.5$	1.18, 38.8	1.10, 47.0
$\mathsf{L} = 64, \lambda = 0.35$	1.15, 41.0	1.09, 53.3
$\mathsf{L} = 64, \lambda = 0.25$	1.12, 52.9	1.07, 58.4
$\mathsf{L} = 64, \lambda = 0.15$	1.08, 61.4	1.06, 70.6
$\mathsf{L} = 64, \lambda = 0.1$	1.07, 74.2	1.05, 91.9
$\mathsf{L} = 32, \lambda = 1$	1.27, 30.0	1.17, 38.6
$\mathsf{L} = 32, \lambda = 0.5$	1.19, 37.9	1.12, 46.9
$\mathsf{L} = 32, \lambda = 0.35$	1.14, 41.3	1.09, 53.3
$\mathsf{L} = 32, \lambda = 0.25$	1.12, 48.7	1.08, 59.8
$\mathsf{L} = 32, \lambda = 0.15$	1.09, 55.5	1.06, 70.7
$\mathsf{L} = 32, \lambda = 0.1$	1.08, 63.7	1.05, 87.9
$\mathsf{L} = 256, \lambda = 1$	1.32, 37.5	1.21, 45.7
$\mathsf{L} = 256, \lambda = 0.5$	1.19, 48.6	1.13, 58.4
$\mathsf{L} = 256, \lambda = 0.35$	1.15, 56.1	1.11, 67.3
$\mathsf{L} = 256, \lambda = 0.25$	1.11, 62.9	1.08, 79.8
$\mathsf{L} = 256, \lambda = 0.15$	1.08, 82.0	1.06, 103.1
$\mathsf{L} = 256, \lambda = 0.1$	1.07, 100.7	1.05, 137.3

Chapter 35

Porting kladde to C++ and into Inkscape

What a second implementation, kladde-cpp, and its first application, the Inkscape fork inkscape-kladde, found out about kladde: where the specification left a second implementer guessing, where C++ made the design harder than Rust does, and what it took to make an existing application durable.

The format held up: kladde-cpp, written from the specification and kladde-rs's store, reads and writes files that kladde-rs reads and writes, in both directions, and the typed layers agree on fingerprints once both name an enum's variants alike. The obstacles were elsewhere. The specification was silent on a dozen points where two implementers could each be right and disagree, which the port turned into rules; C++ cannot enforce the exclusive access that Rust's handles rely on, so the C++ handles carry rules the compiler does not check; and Inkscape needed long-lived handles and a way to observe its document rather than replace it, both of which kladde-cpp now has.

The port's findings are of kladde-rs `5d9ff9f` on its branch `cpp`, kladde-cpp `e8a3e61` and inkscape-kladde `f81602b406`; the measurements are of kladde-cpp `c40ff8f` and inkscape-kladde `6452817ae3` (branch `main`, the typed schema) and `04e6ea0d31` (branch `untyped`, attribute values as text), with kladde-rs `f2f29a7` for the canonical SVG; the commands are at the end.

35.1 What was built

- **The schema layer**, from the specification, checked against the fingerprint vectors.
- **The store**, ported from kladde-rs's `kladde-store` module by module, consolidation included, and tested as kladde-rs tests it: against a plain byte vector after every operation (the differential oracle), with simulated power cuts at every write since the last `fsync`, and with kladde-rs's consolidation thresholds.
- **The typed layer**, header-only, as the draft designed it, with the departures that kladde-cpp's page lists.
- **The Inkscape demo**: documents backed by a kladde file in an XML schema, with no save button.

35.2 Two implementations, one format

Files are format-compatible in both directions, but not byte-identical. Fifty seeds of a random workload with many flushes, consolidation, cuts and pastes, and an unflushed journal, written by one implementation, read the same in the other, which then edits them and hands them back. The bytes differ because kladde-rs iterates hash sets where the order picks consolidation's victims, and kladde-cpp sorts there instead; the fold and consolidation are free to choose, so both files are right.

The typed layers agree on a document only once the C++ type names an enum's variants as the Rust type does. The cross-implementation test writes a drawing — a struct holding text, a vector of shapes, a map, an option and an integer — from each language and reads it in the other. The shapes

are a Rust enum `Shape { Circle { r }, Poly(Vec<Point>), Label(String), Empty }`, and the C++ type `std::variant<Circle, std::vector<Point>, std::string, kladde::Unit>`; thanks to variants that hold one value, the two have the same structure, descriptor for descriptor, and the encodings agree byte for byte. But the fingerprints differed: a variant's name is part of the fingerprint, and a `std::variant` names its alternatives after their types — `Value`, `Text` — where Rust names them `Poly` and `Label`. `KLADDE_VARIANT_NAMED` now gives the C++ type the names the file uses, and with it each implementation opens the other's file.

What it means for the design. Names in the fingerprint are deliberate: evolution reconciles by names, so two types that differ only in names are different types. The cost shows in a language whose sum type has no names of its own, which C++'s `std::variant` is, and will be in Python's unions and TypeScript's too: a program that shares files with another must spell out the other's names. That is a fair price, and kladde-cpp asks it only where a program shares files across languages; within one, the names it derives are stable.

35.3 What the specification left open

Each point below is one where kladde-rs and kladde-cpp, both following the documents, could have disagreed, and most were settled while this was written.

Settled in the specification:

- **A schema table must be canonical to be read.** kladde-rs accepted a varint longer than it needed to be, enum variants in any order (re-sorting them, so that re-encoding changed the bytes) and two variants with one discriminant; kladde-cpp refused all three. Canonical encoding now requires the refusal, so that a table that decodes re-encodes to its own bytes, and kladde-rs refuses them too.
- **An Opaque's parameters are not containment**, so a type may reach itself through one; and **an enum whose every variant holds itself is legal**, if useless (rules of nesting).
- **A `Move` makes no cut** that a `Paste` could name (cuts).

Still open, for the conformance suite to decide:

- whether page 1 of a file that has only ever had one header must be a header page, as the file format says "if present", or may be zeros, as both implementations leave it at creation;
- whether a header whose epoch has the wrong parity for its slot, or whose `journal_pointer` is below 2, makes the file invalid or is ignored, as both implementations ignore it;
- whether varints in journal records and table pages may be longer than they need to be, as both implementations accept;
- whether a trailer that matches its CRC but names page 0, page 1 or a page already in the journal ends the journal, as both implementations treat it, silently dropping what follows, or makes the file invalid;
- whether a transaction of no records is valid, and whether a file with a `format_version` other than 0 is refused.

Where kladde-rs and the reference algorithms disagree, kladde-cpp followed kladde-rs, and the disagreements are now in the implementation notes: the seed of the rotating walk, what the header evicts, who takes the cut's fillers, the order in which the budget loop applies its floors, the depth of the span table, the fold's length check, and the bound on a `Ref`.

35.4 What C++ made harder

The store ported line by line; the typed layer did not. The store is positions and bytes, and the reference algorithms describe it closely enough that a port follows kladde-rs's structure, with one mechanical change: every fallible function returns `kladde::expected<T>`, and `KLADDE_TRY` does what Rust's `?` does. The typed layer is where the languages part.

35.4.1 Handles that nothing checks

Rust's borrow checker enforces two rules that C++ handles can only document.

- *A handle is valid until something is written through a handle that is not itself or made from it.* In Rust a `Mut` borrows its parent exclusively, so a stale one does not compile; in C++ a stale `kladde::Mut` writes at the wrong position. This is the iterator-invalidation rule C++ programmers already keep, and the debug checking the draft proposed — a stack of live handles in the store — is not built yet.
- *A handle must not move while a handle made from it lives*, since the child refers to its parent's place. Rust's borrows pin the parent for free; in C++, `auto [a, b] = c.parts();` is fine and returning a handle by value is fine, by guaranteed copy elision, but moving a handle into a container while its children are in use is not.
- *A cut is pasted at most once.* kladde-rs's `CutId` is neither `Copy` nor `Clone`, so the type system enforces it; kladde-cpp's is a plain struct, and the store's check of it holds only while the cut's journal lasts.

35.4.2 Describing types without a derive

One description per type, written by a macro today and by reflection later, drives everything else, as the draft proposed, and it works. `KLADDE_STRUCT(Contact, email, phones, starred)` expands to a `constexpr` description found by argument-dependent lookup, and to a mixin that gives `Mut<Contact>` the methods `email()`, `phones()` and `starred()`. Two compiler problems cost time, both now in the code as comments:

- a `static constexpr auto` member initialized through argument-dependent lookup inside a constrained partial specialization crashes GCC 13 with an internal error, the compiler that Inkscape builds with on Ubuntu 24.04; a `static constexpr` function avoids it;
- a static member of `durable<T>` cannot be computed by calling `durable<T>`'s own functions, which are not defined until the class is complete, so the sizes and flags that follow from a struct's fields are computed in a helper class of their own.

Recursive types need care that Rust's lazy constant evaluation does not. `kladde::Box<T>` cannot report `T`'s fixed size, since a recursive type reaches itself through a box and asking would instantiate the type inside itself; it reports none, which costs nothing on file and refuses only a `std::vector` of boxes.

35.4.3 Long-lived handles

Inkscape needs handles that live as long as the node they point to, and kladde-cpp's `kladde::List` and `kladde::Ref` give them, as Values and their positions describes. Each element of a list is an *anchor* that knows its list and its index; a list knows the anchor of the value that holds it, and finds its own offset inside that value by walking the value's fields up to itself, each of which knows its encoded size in constant time. So a write through a `Ref` finds its place in `O(depth · log m)`, and its change of size climbs the same path, updating each list's Fenwick tree.

One restriction falls out: **a `List` may sit inside structs, variants, options, boxes and other lists, but not inside a `kladde::Vector`, a `std::vector` or a `kladde::Map`**, whose indexes would not hear of a change made through a `Ref`; the compiler refuses it. A document tree, whose children are lists all the way down, never needs it.

35.5 What Inkscape needed

inkscape-kladde observes Inkscape's XML tree and mirrors every change into the kladde file, rather than replacing the tree's nodes with durable ones, as the draft proposed. The draft's `KladdeNode` would have implemented `Inkscape::XML::Node` over the durable tree, so that the document lives in memory once. But Inkscape's nodes are garbage-collected objects of the class `SimpleNode` and its subclasses, created through the document's factory methods and copied, compared and merged by code that assumes them; replacing them touches the XML layer's every user. A subtree observer on the document node, by contrast, hears every change — attributes, content, insertions, removals, reorderings and renamings — and needs nothing else of Inkscape.

What the observer costs. The document is held twice in memory, once as Inkscape's tree and once as kladde values, and a change is applied twice; the measurements below put a number on it. *What it does not cost:* durability is the same, since each change is recorded before the call that made it returns, and the file is the same, since it holds the same schema either way. The typed SVG schema faces the same choice, and

would face it harder: Inkscape's objects (`SPObject` and its subclasses) are built from the XML tree and keep their own state, so a typed schema would be a third copy, or a rewrite of the object layer.

Three details needed care:

- **Inkscape changes a document's tree while building it**, before anything can observe it: it adds a named view and version attributes as it creates a document from a file. So attaching the observer first compares Inkscape's tree with the file's and writes the difference, and any later event for a node the file does not know rewrites the file from the tree, a fallback that the tests never reach.
- **Saving must not overwrite the kladde file with SVG.** A backed document's filename is its kladde file, so the title bar names it; the commands bar and the File menu offer no *Save*, and Ctrl+S only flushes the file, while *Save As* still writes SVG wherever the user chooses, and an imported `.svg` is left as it was.
- **Only the GUI imports.** Opening an `.svg` in the GUI creates `foo.kladde` next to it, asking first if that file exists; the command line, which Inkscape uses for conversions and exports, opens files as before, so that `inkscape --export-type=png foo.svg` leaves no `foo.kladde` behind.

The idle flush fits Inkscape's main loop: after each recorded change, a low-priority idle callback calls `flush_if_journal_exceeds(0.5)`, at the same priority Inkscape gives its own deferred work, so that a flush, and its `fsync`, falls between gestures rather than in one. The seal and the commit on a worker thread are in kladde-rs and kladde-cpp, off by default, until the demo shows whether the idle flush suffices.

35.6 Measurements

inkscape-kladde costs Inkscape about 5 µs and 19 bytes of journal per edit, and a sixth more memory; it keeps a drawing in a file of 1.2 to 2.2 times its encoding while the drawing is edited, which for the coat of arms and the world map is no larger than the drawing's canonical SVG, and right after the import in a file of 1.03 to 1.4 times its encoding.

`test_kladde-backing-bench` imports three drawings of kladde-rs's corpus — the tiger (67 KB of SVG), the coat of arms of Spain (193 KB) and the world map (1.1 MB) — and makes a drawing program's edits through Inkscape's XML tree: drags of twenty moves of one element (eight in ten), style changes, reorders, duplicates and deletions. It runs the same edits on the same drawing without a backing, for what Inkscape spends on them itself, and checks that the closed file reopens as the document was. The fork stores a drawing in a typed schema, on its branch `main`: each element's attributes are one ordered list of entries, each a typed value — lengths, paints, colors with their written form, transform lists, path data with its layout, points, style declarations, decimal numbers, embedded data as its bytes — or the attribute's text, where formatting the typed value would not reproduce Inkscape's text byte for byte; 98.4 to 99.8% of the drawings' attributes end up typed. The numbers are the typed schema's, at the journal budget sized to the file that kladde uses by default, unless they say otherwise; the XML schema of the branch `untyped`, which keeps every attribute value as text, serves as an ablation. Each number is one run on a shared machine of four cores, and a range spans a run of 10,000 edits and one of 50,000; the heap is `mallinfo2`'s, before and after.

drawing	SVG	document's heap	backing's heap	edit, plain	edit, backed	journal per edit	file after the import (encoding)	closed after 10,000 / 50,000 edits (encoding)
tiger	67 KB	4.2 MB	0.64 MB	2.0–3.0 µs	7.4–7.6 µs	19 B	60 KB (42 KB)	112 / 136 KB (52 / 65 KB)
coat of arms	193 KB	5.9 MB	1.12 MB	2.1–2.2 µs	7.8–7.9 µs	20 B	112 KB (96 KB)	196 / 284 KB (113 / 149 KB)

drawing	SVG	document's heap	backing's heap	edit, plain	edit, backed	journal per edit	file after the import (encoding)	closed after 10,000 / 50,000 edits (encoding)
world map	1.1 MB	29 MB	4.4 MB	2.1 µs	8.0–10.5 µs	19 B	612 KB (594 KB)	712 / 900 KB (604 / 642 KB)

An edit costs about 5 µs more, and stays that cheap however long the journal grows. The journal is appended without an `fsync`, and the flush that folds it, once every 400 to 500 edits, is amortized over them; the world map's 10.5 µs came from a run of 10,000 edits whose slowest move took 23 ms, an outlier its run of 50,000 did not repeat. An early version spent time quadratic in the journal's length, since each append copied the journal's list of cuts, which every attribute change extends; both implementations now append in constant time. Saving the whole drawing as SVG, which an application without kladde does on every save, takes 1.2 to 12.7 ms here, and writes the whole file.

The second copy of the document costs a sixth of what Inkscape holds for it: 0.64 MB beside the tiger's 4.2 MB, and 4.4 MB beside the world map's 29 MB, since Inkscape's tree, its objects, their styles and their rendering structures dwarf an encoding that holds each node's name, attributes and children once.

Payload pages halve the file right after the import. Importing is one splice of the whole encoding, whose bytes payload pages write once, into pages that the flush adopts as data pages. Without them, the bytes went into the journal first and then into data pages, and the journal's pages became holes below them: kladde-cpp `c4ebcb2` left 100 KB, 204 KB and 1204 KB for the same three imports, where `c40ff8f` leaves 60 KB, 112 KB and 612 KB. After thousands of edits the difference is gone: closed after 10,000 edits, the files were 112 KB, 200 KB and 696 KB without payload pages, since consolidation drains the import's holes along the way.

While a drawing is edited, its file stays within 1.2 to 2.2 times its encoding, which holds the journal, the pages a flush keeps for the header before it, and those a flush frees but can reuse only after the next one; closing flushes and truncates, so a closed file is about as large as the open one was. The drawing's canonical SVG is the drawing as the edits leave it, saved by Inkscape and written again by kladde-svg with less whitespace, shortest decimals and shortened path data; it takes 89 / 115 KB, 212 / 286 KB and 1107 / 1192 KB after 10,000 / 50,000 edits. Next to it, the kladde file is 1.2 to 1.3 times as large for the tiger, whose fixed overhead weighs most beside its small content, 0.9 to 1.0 times for the coat of arms, and 0.6 to 0.8 times for the world map, whose numbers and path data the typed schema stores more compactly than their text.

Opening a kladde file takes 1.3 to 2.3 times as long as parsing the SVG: 17 to 21 ms for the tiger against 11 to 12 ms to parse it, 31 to 46 ms for the coat of arms against 19 to 20 ms, and 127 to 147 ms for the world map against 93 to 99 ms. Most of the time goes to building Inkscape's document from the tree, as from a parsed SVG; formatting the typed values back into Inkscape's text adds a little, since with values kept as text a file opens in 0.9 to 1.8 times the parsing time.

35.6.1 What the typed schema took

What the typed schema needed, beyond keeping attribute values as text, says more than its numbers:

- **An existing application's formatting must be stored as data.** kladde-svg normalizes what it parses, and would have lost nearly everything Inkscape writes; reproducing it took decimal numbers rather than floats, a color's written form, a transform's optional arguments, and how a path's text separates its numbers — and Inkscape's own `10e-4` and `-0` still fall back to text.
- **A `Mut`'s whole-value `set` records the whole encoding**, where `std::string`'s records only the changed middle, so changing one number of a long path rewrote the path; the branch diffs the encodings itself, and kladde-cpp should offer it for every type, as the specification guarantees for Text and Opaque.

- **A `std::variant` with many alternatives of one type** — `x`, `y`, `width` and twenty more are all lengths — cannot be reached through `if_holds<A>`, which needs an index-based form, and its names must be written out alike.
- **A `std::vector` of varint numbers is refused**, since they are not fixed-size, so short lists that are only ever replaced whole carry a `kladde::Vector`'s index.
- **Adding a typed attribute changes the root fingerprint**, so that files written before no longer open; growing a typed schema over time needs evolution, which neither implementation has yet.
- **Embedded data is held as its bytes**: a `data:` URI with base64 keeps its media type and decoded bytes, three quarters of its text, and only the line breaks inside the base64 do not read back; none of the three drawings embeds data.
- **Under the observer, a typed copy is a third copy of the document**, beside Inkscape's XML strings and its objects; a typed schema pays off fully only where it replaces the application's own model, which in Inkscape would mean rewriting its object layer.

35.6.2 The journal's budget

kladde sizes the journal's budget to the file, which keeps these files within 1.2 to 2.2 times their encoding at a flush of about a millisecond every 400 to 500 edits; a fixed budget of 256 pages, which almost never flushes, lets the tiger's file grow to 16 times its encoding. The budget is 2% of the sequence in whole pages, at least 2 and at most 10 (`Options::journal_budget_pages` left empty, in both implementations), which gives the tiger and the coat of arms 2 pages and the world map 3. The runs below are the benchmark above, with `KLADDE_JOURNAL_BUDGET_PAGES` set for the fixed budgets; the ratios are to the encoding after the edits, and the latencies are those of each single move of a drag, which in the benchmark includes any flush the move triggers, since nothing flushes at idle there.

budget	drawing	closed after 10,000 / 50,000 edits	flushes per 10,000 edits	mean edit	drag move, 99.9th percentile	slowest move
sized to the file (2 pages)	tiger	112 KB (2.2×) / 136 KB (2.1×)	23	7.4–7.6 µs	1.0–1.1 ms	3.2 ms
sized to the file (2 pages)	coat of arms	196 KB (1.7×) / 284 KB (1.9×)	25	7.8–7.9 µs	1.1 ms	4.3 ms
sized to the file (3 pages)	world map	712 KB (1.2×) / 900 KB (1.4×)	20	8.0–10.5 µs	1.3–1.7 ms	23 ms
1 page	tiger	108 KB (2.1×) / 140 KB (2.2×)	47	9.1–10.5 µs	0.9–1.1 ms	3.9 ms
1 page	coat of arms	220 KB (1.9×) / 268 KB (1.8×)	50	10.9–12.1 µs	1.1–1.2 ms	4.2 ms
1 page	world map	744 KB (1.2×) / 896 KB (1.4×)	48	9.9–10.7 µs	1.0–1.2 ms	4.3 ms
4 pages	tiger	128 KB (2.5×) / 156 KB (2.4×)	11	7.4–9.8 µs	1.5–1.6 ms	4.0 ms
4 pages	coat of arms	188 KB (1.7×) / 288 KB (1.9×)	12	7.5–8.1 µs	0.4–1.6 ms	4.1 ms
4 pages	world map	692 KB (1.1×) / 888 KB (1.4×)	11	7.2–7.7 µs	0.1–1.9 ms	8.8 ms

budget	drawing	closed after 10,000 / 50,000 edits	flushes per 10,000 edits	mean edit	drag move, 99.9th percentile	slowest move
256 pages	tiger	272 KB (5.2×) / 1040 KB (16×)	0	5.2–6.2 µs	46–54 µs	0.9 ms
256 pages	coat of arms	340 KB (3.0×) / 1144 KB (7.7×)	0	5.0–5.2 µs	35–36 µs	4.2 ms
256 pages	world map	836 KB (1.4×) / 1660 KB (2.6×)	0	5.1–6.4 µs	42–49 µs	3.5 ms

The world map's slowest move of 23 ms at the budget sized to the file came from its run of 10,000 edits, whose run of 50,000 peaked at 3.0 ms. At 256 pages, the 50,000 edits never filled the journal, which is why no move stalled for long there; the next edit would have folded a journal of a megabyte, as the ablation shows, whose larger records fill it once in 50,000 edits.

The budget trades a few long stalls for many short ones, and the short ones are the better deal for an editor. A median move costs 3.1 µs at every budget: the move itself, recorded in the journal. With 256 pages, a move almost never meets a flush, but the one that does folds a journal of a megabyte and stalls for 46 to 77 ms, three to five frames; with 4 pages, one move in seven hundred meets a flush of a few pages, which takes one to two milliseconds and fits within a frame. The flushes here run inside the moves because the benchmark has no idle time; in the GUI, the idle flush moves most of them between gestures, and does so better the smaller the journal it has to fold. The times come from a disk whose `fsync` is fast; on a slow disk, every flush costs more, and the small budget's frequent flushes cost proportionally more than its few large ones save, which is the case for the commit on a worker thread.

Hence a budget that follows the file, between a floor and a ceiling, rather than a fixed one. A budget of 1 page gains nothing in size over 2 to 4 here, and flushes two to four times as often; above a few pages, the files grow with the journal and its fallback while the flushes saved bring little, since each flush is also a chance to consolidate. Transactions and batches gives the rule and its reasons.

35.6.3 Ablation: attribute values as text

Keeping every attribute value as text, as the branch `untyped` does, makes the encoding 1.5 to 1.8 times larger, the files 1.5 to 1.7 times larger and an edit's journal a fifth larger; in exchange it takes two thirds of the typed schema's memory and a third to a half of its import time, and opens a little faster. The branch is `main` with the XML schema in place of the typed one, and the runs are the same.

	tiger	coat of arms	world map
encoding, typed / text	42 / 64 KB	96 / 172 KB	594 / 1071 KB
backing's heap, typed / text	0.64 / 0.41 MB	1.12 / 0.77 MB	4.4 / 2.7 MB
import, typed / text	6.7 / 3.1 ms	8.8–8.9 / 3.4–4.5 ms	44–49 / 15–19 ms
opening, typed / text (parsing the SVG)	17–21 / 15–20 ms (11–17 ms)	31–46 / 28–31 ms (18–20 ms)	127–147 / 112–125 ms (93–122 ms)
journal per edit, typed / text	19 / 23–24 B	20 / 23–24 B	19 / 23–24 B
mean edit, typed / text	7.4–7.6 / 7.1 µs	7.8–7.9 / 7.2–7.4 µs	8.0–10.5 / 6.4–7.2 µs
file after the import, typed / text	60 / 80 KB	112 / 188 KB	612 / 1092 KB

	tiger	coat of arms	world map
closed after 10,000 / 50,000 edits, typed	112 / 136 KB	196 / 284 KB	712 / 900 KB
closed after 10,000 / 50,000 edits, text	164 / 232 KB	300 / 460 KB	1192 / 1544 KB
canonical SVG after 10,000 / 50,000 edits	89 / 115 KB	212 / 286 KB	1107 / 1192 KB

With values as text, a kladde file is 1.1 to 2.0 times the canonical SVG, where the typed schema's is 0.6 to 1.3 times it: both hold the drawing's numbers, but as text they keep Inkscape's whitespace and digits, which the canonical SVG drops and the typed schema never stores. Per edit, the two cost the same within the noise of the machine, though a median move costs 3.1 µs typed against 2.6 µs as text, which is parsing and formatting the attribute's value. The journal budget behaves as with the typed schema, the world map getting 5 pages for its larger encoding: at the budget sized to the file the files stay within 1.1 to 2.1 times their encoding, at a flush every 300 to 900 edits. At 256 pages its larger records fill the journal once in 50,000 edits, and the move that meets that flush stalls for 46 to 77 ms while it folds a megabyte of journal, where no move at the budget sized to the file took longer than 4.1 ms.

35.6.4 In the GUI

In the GUI, under a virtual display (inkscape-kladde `f81602b406`, with the XML schema), opening `tiger.svg` created `tiger.kladde` beside it and left the SVG as it was; flipping and rotating the whole drawing through Inkscape's actions, then killing Inkscape without saving, left a file that the command line exports with the flip and the rotation in place. The *File* ▸ *New* dialog, which asks where the new document's file goes, was not exercised, since no test drives a file chooser.

35.7 Bugs the port found in kladde-rs

The port read kladde-rs's store line by line, and found five things that are wrong rather than different; the implementation notes record each.

- ⚠ **A single write of 4 GiB or more corrupts the journal**, since a record's length is kept as a `u32`.
- **Arithmetic on values read from the file is unchecked**, so a file whose CRCs match but whose numbers do not panics or wraps rather than being refused.
- **`FileStorage` takes no lock**, so two processes can open one file.
- **A cut that escapes a transaction rolled back by `Store::atomically` can paste another record's bytes.**
- **`TypeTable::fixed_size` shared one memo between plain sizes and fixed forms**, so a type met inside a `Fixed` and outside one got one size for both; fixed in `d19a613`.
- **The first version of the seal could replay a lost journal after a power cut**, since the page it named was cleared without an `fsync` before the seal was appended; porting it found this, and both implementations and the specification now sync first.
- **An append copied the journal's list of cuts**, making a journal quadratic in its length, as the measurements found; fixing it took kladde-rs's `svg-bench` from 136,000 to 330,000 edits a second on the tiger, and from 90,000 to 238,000 on the coat of arms.

And it confirmed one already known: ⚠ a file whose span table outgrows the header's references to leaves, at some two million spans, no longer opens, in kladde-rs and in kladde-cpp alike.

35.8 Reproducing

```
# kladde-cpp: the three test suites, under AddressSanitizer and UBSan
cmake -S kladde-cpp -B kladde-cpp/build -G Ninja -DCMAKE_BUILD_TYPE=Debug -DKLADDE_SANITIZE=ON
cmake --build kladde-cpp/build
kladde-cpp/build/kladde-schema-tests && kladde-cpp/build/kladde-store-tests && kladde-cpp/build/kladde
```

```
# The typed cross-implementation test, in both directions
cargo run -p kladde-types --example cross_impl -- write rs.kladde
kladde-cpp/build/kladde-cross write cpp.kladde
kladde-cpp/build/kladde-cross check rs.kladde
cargo run -p kladde-types --example cross_impl -- check cpp.kladde

# The Inkscape demo's tests and measurements, in inkscape-kladde's build directory
ninja test_kladde-backing test_kladde-backing-bench
./bin/test_kladde-backing
# (created_KB is the file right after the import, file_KB and closed_KB the file before and after clos
KLADDE_BENCH_STEPS=50000 KLADDE_SVG_CORPUS=…/kladde-rs/crates/kladde-svg/corpus/drawings ./bin/test_kl
# … with a fixed budget, and keeping the edited drawings for their canonical SVG's size (kladde-rs's s
KLADDE_JOURNAL_BUDGET_PAGES=4 KLADDE_BENCH_SAVE_DIR=edited KLADDE_SVG_CORPUS=… ./bin/test_kladde-backi
cargo run --release -p kladde-svg --bin svg-roundtrip -- --csv canon.csv edited
```

Chapter 36

Related work

What applications use in kladde's place, where kladde's design comes from, and where it deliberately parts company.

36.1 Systems that address related problems

This section discusses systems that are related to kladde *from the perspective of a user*, addressing questions like "how is kladde different from SQLite, or from whole-file serialization"?

Applications have kept their documents safe from crashes in six ways, and each of the five in use today gives up at least one of three things that kladde combines: one model, the application's own values in memory, which are what the file holds; every change durable against an application crash when its call returns; and writes in proportion to the change, rather than to the document or to the value the change lands in. Saving the whole document gives up the second and the third. A journal beside a checkpoint of the whole state keeps both between checkpoints, and gives up the third at each checkpoint. An embedded database such as SQLite gives up the first, and the third below a page, or below a whole value whose size changes. An object store keeps the first, but writes only at a commit that the application issues, and often each changed object whole. A CRDT library keeps the third, but replaces the application's types with a JSON-like model and a history meant for merging. Durable containers in persistent memory kept all three, and needed hardware that is no longer made. Each approach is still the better choice where an application needs what it offers instead — a format that other programs read, queries, data larger than memory, several processes on one file, or collaboration — none of which kladde offers.

approach	in industry	gives up	does better than kladde
saving the whole document	most document applications; macOS's autosave; Word's AutoRecover; Android's DataStore	each change's durability; writes in proportion to the change	a format other programs read; any type; the simplest code
a journal beside a checkpoint	Figma's servers; Chromium's sessions; Prevayler; LMAX; Tencent's MMKV	writes in proportion to the change, at each checkpoint	a log of the application's own commands
an embedded database	SQLite, alone or under Core Data, SwiftData or Room; LMDB, RocksDB	one model; writes below a page, or below a value whose size changes	partial loading, queries, several processes, maturity
an object store	ZODB; EclipseStore; Realm	durability without a commit; often writes below an object	partial loading, data larger than memory

approach	in industry	gives up	does better than kladde
a CRDT library	Automerge; Yjs	the application's types; a file without history	collaboration and sync
durable containers	Intel's PMDK; Corundum	ordinary hardware	no file to load at open

36.1.1 Saving the whole document

Most applications still write their whole model at each save, made safe by writing a new file and moving it into place; a crash loses everything since the last save, and each save costs time and writes in proportion to the document. macOS's `NSDocument` can autosave in place since 10.7 (2011), by writing a new file and moving it over the document; Apple's guide notes that "the system does not attempt to save every change immediately", and warns that a slow save "could cause periodic blocking of your user interface". Microsoft Word keeps an AutoRecover copy of each open document, written every ten minutes by default and removed when the user saves. Android's Jetpack DataStore persists one typed object, in Protocol Buffers or JSON, by an atomic read-modify-write of the whole, and calls itself "ideal for small datasets"; it "does not support partial updates", and for anything larger it points to Room, an object mapping over SQLite. Firefox writes its whole session every 15 seconds, which users measured in 2016 at 10 GB a day and more. On the drawings and texts that kladde is measured on, saving the SVG every 100 edits would write 590 to 10,699 bytes per edit, and saving the text every 1000 keystrokes 241 and 700 bytes per keystroke, where kladde writes 112 to 389 and 53 to 58 (measured).

What a whole-document save keeps in exchange is real: the file is in a format that other programs read, whatever the serializer handles can be saved, and no type has to change.

36.1.2 A journal beside a checkpoint of the whole state

Systems that cannot afford to lose a minute of work hold their state in memory, append each change to a journal, and now and then write a checkpoint of the whole state that replaces the journal: kladde's own shape, built anew for each application, and with checkpoints that cost as much as the state. Figma's multiplayer servers hold each open file in memory and checkpoint the whole file every 30 to 60 seconds; since 2022 they also append the changes to a journal about every half second, and recover by replaying it over the newest checkpoint, so that "95% of edits to a Figma file get saved within 600ms". Chromium's session service appended commands to its session file and recreated the file every 250 commands. Prevayler (2001) does this for any Java object graph: every change is a `Transaction` object, journaled and synced before it runs, and deterministic, so that replaying it reproduces it — it may not even read the clock. The LMAX exchange runs its business logic in memory on the same principle, journaling its input events and recovering from snapshots (Fowler, 2011). Tencent's MMKV, the key-value store of WeChat, holds its map in memory and appends each write, encoded as a protobuf, to a memory-mapped file, so that the write survives an application crash at once; when the file is full, MMKV rewrites it with only the newest value of each key, and doubles it if that is not enough.

Their journals hold what only the application can replay, and their checkpoints write everything. Figma's changes, Chromium's commands and Prevayler's transactions mean something only to the application's code, which recovery has to run, and run deterministically. kladde's journal holds byte effects on the encoding of typed values, which recovery replays without the application or its schema. A checkpoint of the whole state takes as long as the state, however little changed since the last one, where kladde's flush writes what changed. And MMKV promises nothing under a power cut.

36.1.3 Embedded databases: SQLite and its kin

SQLite is the strongest alternative, and argues it for itself: an application file in SQLite is written only where it changed, as the changes occur, and safely — but the application has to map its model to tables and keep the two in step. SQLite's page on application file formats states that "only those parts of the file that actually change are written out to disk", that "changes can be written back to the disk as they occur" instead of at *File/Save*, and that an undo stack kept by triggers in the file lets "undo/redo ... occur across session boundaries". In WAL mode with `synchronous=NORMAL`, it offers the same two tiers

as kladde's journal and flush: "Transactions are durable across application crashes", while a power cut may roll back the transactions since the last checkpoint but cannot corrupt the file. Its session extension records what transactions changed as a *changeset*, which can be inverted to undo them, concatenated with others, and applied to another copy of the file. It reads only what a query asks for, so a document need not fit in memory, and "the application can do queries to extract only the information needed for the initial screen"; several processes can read and write one file; its format has stayed compatible since 2004; the US Library of Congress recommends it for preservation; and it is "robust against maliciously malformed database files".

SQLite writes in units of pages, and within a row in units of whole values whose size changes. Each commit in WAL mode appends every page it changed to the log, at least one page of 4 KiB by default, where a kladde transaction for an edit of a drawing appends 31 to 43 bytes to the journal (measured). An `UPDATE` overwrites a row in place only if its size stays the same (since 2018), and otherwise writes the row again, overflow pages included; incremental BLOB I/O overwrites bytes in place, but "it is not possible to increase the size of a BLOB". So an application that keeps a long text in one row writes all of it at each typed character, and avoids that only by splitting the text into rows of paragraphs or lines, which it then has to keep in order itself. Apple Notes keeps each note's body as a gzipped protobuf in one column of its SQLite file: SQLite holds the notes, and each note is still written whole.

The mapping is the cost that kladde avoids. In tables, a tree becomes rows with parent keys, ordered children need a position column that an insertion renumbers, and variants become separate tables or nullable columns. An application that shows its whole document keeps a model in memory anyway, and has to issue, for each change to it, the SQL that mirrors the change; a statement it forgets lets the file drift from the model unnoticed. Object mappers take part of this on, at the price of a model that fits tables: Apple's Core Data, and SwiftData, which saves a context's changes at `save()` or at an autosave whose moment Apple leaves unspecified; and Android's Room. Embedded key-value stores such as LMDB, RocksDB and LevelDB sit one level lower: atomic transactions over byte strings, with serialization and the design of keys left to the application.

36.1.4 Object stores

Object stores persist the application's objects themselves, as kladde does, but write a changed object only at a commit that the application issues, often whole, and load objects lazily, so that a document need not fit in memory. ZODB, the object database under Zope and Plone, notices assignments to attributes of `Persistent` objects, but not changes inside a plain `list` or `dict`, for which it offers `PersistentList`, `PersistentMapping` and B-trees: kladde's need for containers of its own, met the same way. Each persistent object is "stored in its own database record", written whole at `transaction.commit()`, and loaded when first accessed. EclipseStore, formerly MicroStream, persists plain Java object graphs: the application calls `store` on each object it modified, and since storing skips objects already stored, a field changed in place goes unwritten unless its own object is stored too. Its storage is log-structured like kladde's: each store is appended to the newest file, a background housekeeping rewrites files whose live share falls below 75 %, and a garbage collection removes entities that the root no longer reaches. Realm, for mobile applications in Swift, Kotlin, .NET, JavaScript and Dart, reads objects through accessors into its memory-mapped file and writes them in write transactions, one at a time per file; MongoDB deprecated Realm's SDKs and its sync service in September 2024, and the on-device database lives on as open source. Persistence goes furthest in MUMPS, the language of Epic's health records: any variable whose name begins with `^` is a *global*, a sparse tree of strings and numbers that lives on disk and is shared by every process.

What sets kladde apart from these stores is that a mutation needs no commit to be durable, and that a change inside a long string or list writes the bytes that changed rather than the object that holds them. What sets them apart from kladde is that they read only what is accessed.

36.1.5 Collaborative documents: CRDTs

CRDT libraries record each change as it happens and save incrementally, as kladde does, for a model built for many writers: a JSON-like document that keeps the history it needs to merge, stored wherever and whenever the application chooses. Automerge changes a document in a callback; `saveIncremental` returns the changes since its last call, to be appended to a file, and `save` writes the whole document compactly; automerge-repo writes each change to its storage as it happens, and now and then

compacts the changes into a snapshot. Yjs works the same way, and Apple Notes keeps a CRDT inside each note to merge edits from several devices. MobX-State-Tree, a state container for JavaScript applications, intercepts mutations as kladde does, but with no file behind them: it emits a JSON patch for each mutation and a snapshot at the end of each action, and leaves storing either to the application. An application that needs collaboration or sync is better served by a CRDT, since kladde merges nothing; one that does not would pay for a history it never uses, and give up typed values.

36.1.6 Durable containers in persistent memory

The API closest to kladde's was built for byte-addressable persistent memory — containers whose updates inside a transaction are durable, with no file format between them and the program — and it went with the hardware. Intel's PMDK offered, in `libpmemobj-cpp`, a persistent `vector`, `string` and `concurrent_hash_map`, updated in transactions on Optane memory. Corundum (ASPLOS 2021) brought the model to Rust, and used its type system to rule out, at compile time, pointers from persistent into volatile memory and updates that bypass the log, the second of which kladde-rs's `Mut` handles prevent as well. Intel wound down PMDK along with Optane in 2022, and archived `libpmemobj-cpp` in March 2023. kladde offers the same programming model on ordinary files, where durability comes from `fsync` and copy-on-write rather than from flushing cache lines.

36.1.7 What they offer that kladde does not

Five features of these systems have no counterpart in kladde yet.

- **Holding back what others see until it is durable.** Cloudflare's Durable Objects let a write return at once, into an in-memory cache, and hold back every outgoing message until the write is durable; should the write fail, the messages turn into errors and the object restarts. A kladde mutation, too, returns before it is safe from a power cut, and an application that tells someone else about a change — a server, another process, the user through a "saved" indicator — has to remember to call `flush()` first.
- **Changesets.** SQLite's session extension turns transactions into changesets that can be inverted, concatenated and applied to another copy, which is what undo, redo and synchronization between copies are built from; kladde's journal holds the same information, at the level of bytes, until the next flush folds it.
- **Partial loading.** SQLite, ZODB, EclipseStore and Realm read what is accessed; kladde reads the whole file at open, and segments are planned for this.
- **Readers in other processes.** SQLite, Realm and MMKV let other processes read a file while one writes; kladde's scope excludes them.
- **A stance on hostile files.** SQLite promises that a malformed file cannot provoke a memory error in its reader; kladde's checksums and fingerprints guard against accident, not against an adversary, and its specification says nothing about files built to attack a reader — though a document file is exactly what gets sent by email.

36.2 Systems with related internal parts

This section discusses systems that overlap with kladde in terms of specific internal parts — often, these are systems from which kladde borrows ideas to solve some common subproblem in a different context.

Measured against what applications use in its place, kladde combines three things, and every approach in use today gives up at least one of them: one model, the application's own values in memory, which are what the file holds; every change durable against an application crash when its call returns; and writes in proportion to the change. What applications use instead compares kladde with whole-document saves, journals beside checkpoints, embedded databases such as SQLite, object stores, CRDT libraries and durable containers in persistent memory; the sections before it compare kladde's parts with the systems they come from.

No system known to these documents combines what kladde does, but each of its layers has a close precedent: kladde is an editor's piece table kept on disk, linked rather than keyed by position, made crash-safe by shadow paging with an intent log, kept compact by a log-structured cleaner, and holding a serialization much like Avro's. None of the load-bearing pieces is novel, which is the point: 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.

layer	closest precedents	shared	different
the encoding	Avro's binary encoding	the schema in the file, no tags, no padding, varints, counts	edited in place; text ends in a terminator; `Fixed`
the sequence on file	Word's fast save; FAT's chains	an on-disk piece table; successors in a table	links instead of positions; byte-granular spans
the span table	LFS's inode map; Bitcask	a table of locations, newest wins, held in memory	nothing is searched on disk
the header as write buffer	LSM trees' memtables	hot entries buffered, cold ones evicted to runs	one tier of unordered leaves, no levels
commit and recovery	shadow paging; LMDB; WAFL and ZFS	a root written last after an `fsync`; a log replayed over it	no tree on disk; one `fsync` per commit
reclamation	Sprite LFS's cleaner; SSD flash translation layers	victims by utilization and age	pages instead of segments
state and durability	system prevalence; Redis	all state in memory, every change logged	byte records instead of commands; incremental checkpoints
moving values	Cap'n Proto's orphans	detach, adopt elsewhere, no copy	journaled, within a transaction
positions in memory	ropes and piece trees	a counted tree over pieces	—
order by links	Yjs	a linked list of items with ids, merged into runs	one writer: no merging, tombstones or history

36.2.1 The encoding: Avro's, nearly

Avro's binary encoding is the closest precedent of the sequence's encoding, and its object container files carry their schema as kladde files do. An Avro record is its fields' encodings concatenated, with no tags and no padding; integers are zigzag varints, a union starts with its branch's index as a varint, and an array is its elements in blocks, each preceded by its count. kladde's evolution takes Avro's model of reconciling a writer's schema with a reader's as well. The differences follow from mutation: Avro data is written once and read in sequence, while kladde edits values in place, so it keeps counts in one place rather than in blocks, ends text in a terminator so that an edit of text needs one splice, and offers `Fixed` for values that change often. *Protocol Buffers* share the varints and drop padding too, but tag every field with its number, which makes evolution possible without a stored schema at the price of bytes per field.

36.2.2 The sequence on file: a piece table, as Word's fast save kept one

A piece table on disk is old: Microsoft Word's binary `.doc` format kept one for its *fast save*, which appended the changed text to the document's stream and wrote a new piece table over it. Bravo, at Xerox PARC in the 1970s, introduced piece tables for editing, and Visual Studio Code's piece tree (2018)

keeps one in a balanced tree in memory, as the rope does here. Word's table differs from the span table in three ways that matter. It keys each piece by its character position, so it is rewritten whole at every fast save, where links let kladde state only the spans at an edit. It reclaims nothing until a full save rewrites the document, so deleted text stayed in fast-saved files, which leaked it to whoever received them. And it gave no crash safety of its own.

The span table's successors are *linked allocation*, as FAT keeps it, with byte-granular spans instead of clusters. FAT file systems, and Microsoft's compound files, keep each file or stream as a chain of fixed-size clusters, whose successors a table holds. A chain of fixed-size clusters can take an insertion only at a cluster's boundary; a span can be split at any byte. And linked allocation's classic weakness, reaching a byte by walking the chain to it, does not arise, since kladde walks each chain once, at open, and keeps a rope. The other classic scheme is *extents* keyed by their offset in the file, as ext4's and XFS's extent trees keep them; inserting a range there, as Linux's `FALLOC_FL_INSERT_RANGE` does, shifts every extent behind it — the cost that keying spans by id avoids, and that kladde's previous design paid per allocation. The name *extent* would mislead for that reason, and *span* is used instead; the fold's transient runs keep the literature's name, *pieces*, and a span is a piece made persistent: it has an id, links to its successor, and outlives the flush.

36.2.3 Storage: a log-structured file system for one file

The pages, the span table, the cleaner and the two header slots are those of a log-structured file system, with pages for segments and one file in it. Sprite LFS (Rosenblum and Ousterhout, "The Design and Implementation of a Log-Structured File System", SOSP 1991, and TOCS 1992) writes everything to fresh segments, finds each file's inode through an *inode map* that is itself written to the log and held in memory, tracks each segment's live bytes in a *segment usage table*, cleans by moving live blocks out of segments chosen by utilization and age, and alternates between two checkpoint regions. The span table is that inode map, coverage the segment usage table, cost-benefit scoring its cleaning policy, and the header slots the checkpoint regions. Its cleaning unit is a large fixed segment; kladde's is a page, and its spans can be re-cut freely because the directory keeps no order. *Bitcask* (Basho, 2010) holds every key in memory, rebuilt at open from its log, lets the newest write of a key win, and merges files by writing their live values again: the span table, read at open, is a Bitcask keyed by span id. *LSM trees* (LevelDB, RocksDB) resolve a key's versions across runs by recency, as entries are resolved across epochs, and the header as write buffer is a memtable whose eviction is an L0 flush; but LSMs keep runs sorted and levelled so point lookups work without full residency, while kladde is fully resident after the bulk read and needs no sort order, no levels and no bloom filters.

The commit is shadow paging with an intent log. System R's shadow pages (Lorie, "Physical Integrity in a Large Segmented Database", TODS 1977), LMDB's two meta pages, WAFL's consistency points with their log of requests in NVRAM (Hitz, Lau and Malcolm, USENIX Winter 1994), and ZFS's transaction groups with their uberblocks and intent log all write a new root last, after an `fsync`, over pages that no older root reaches, and replay a log over the last root after a crash. kladde's flush is such a consistency point, its journal such an intent log, and its two header slots LMDB's meta pages; it has no tree on disk to copy paths in, since nothing is searched on disk. 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. LMDB also needs two `fsync`s per commit where kladde needs one, because kladde can let the header lag one write behind and lean on journal replay. btrfs's *inline extents* and ZFS's *bonus buffers* store small data directly in metadata — the precedent for `Inline` entries.

36.2.4 State and durability: a memory image with incremental checkpoints

Keeping all state in memory, logging every change before it returns, and folding the log into a checkpoint is *system prevalence*, as Prevayler implements it and Martin Fowler describes it as a *memory image*; Redis does the same with its append-only file and its snapshots. kladde differs in two ways. Its log holds byte effects rather than commands, so replaying it needs no application code and no schema; and its checkpoints are incremental, since a flush writes what changed, where those systems write whole snapshots or rewrite their whole log.

36.2.5 Moving values: Cap'n Proto's orphans

Cut and paste **is Cap'n Proto's *orphans*: an object disowned by its parent and adopted by another in the same message, without copying, move-only, and zeroed if nothing adopts it.** Here the cut lives in the journal's transaction rather than in a message's arena, and the flush turns it into a few links.

36.2.6 Order by links: sequence CRDTs

Yjs **keeps a shared text as a linked list of items, each with an id and links to its neighbours, merges consecutive items of one writer into runs, and persists updates that it later merges into one, which is close to linked spans with ids.** *Automerge* keeps the history of operations, in a compressed, columnar format. Both merge concurrent edits from many writers, and so keep tombstones and history and compact by encoding the whole document again; a single writer needs none of that.

36.2.7 Where the redesigns have led

Each redesign moved kladde further from object stores and closer to document storage.

1. **The first design** was a relocatable heap: allocations contiguous in the file, pointers between them, and compaction that moved whole allocations — a persistent heap, as PMDK's `libpmemobj` or Boost.Interprocess's mapped files keep one, and an object store, as ZODB's `FileStorage` or Realm are.
2. **The second** kept allocations and pointers, but stored them as a log-structured file system stores files: pages written copy-on-write, an address table of extents keyed by offset, an intent log, and a cleaner.
3. **The present design** keeps the file system and drops the objects: one file whose extents are linked, a piece table, holding a serialization as Avro would write it.

So kladde has converged on known parts, put together in a way not found elsewhere: a typed document held whole in memory, durable at every change through a byte-level journal, and saved as deltas into a linked piece table on copy-on-write pages. The nearest whole is Word's fast save, without its rewrites of the piece table, its leaked deletions, and its lack of crash safety.

One lesson taken from the same literature rather than from a system: PostgreSQL's transaction-id wraparound "freezing" is why epochs are 64-bit.

Chapter 37

Implementation notes

What implementing the design in kladde-rs has shown: where kladde-rs falls short of these documents, and what it chose where they leave a question open.

This page is where kladde-rs's findings are recorded as they come up, and proposed for the other documents from here. Paths like `spec/journal.md` are relative to `content/`, the root of these documents. Branches of kladde-rs that hold earlier designs, such as `address-table`, keep their own `implementation-notes.md`.

37.1 Where the implementation falls short

What the code does not do yet, or does less well than kladde-docs or its own goals ask, and what that costs.

- ⚠ **The span table has two levels only: the header, and the leaves it names.** A leaf that names table pages of its own is refused at open. The header's room for references to leaves, one or two bytes each, bounds a file to some 2,000 to 4,000 leaves, one to two million spans: a sequence of a few hundred MB in spans of the length the measurements find, and far more than any workload measured, the largest of which takes 21 to 26 leaves. Beyond the bound, a flush fails with `Error::OutOfBounds`, and since opening flushes the replayed journal, a file whose journal holds such a flush no longer opens. No workload measured comes near the bound: the world map's sequence of 942 KiB takes 21 to 26 leaves. The fix: write interior table pages when the leaves' references outgrow the header, with the path copy of `impl/consolidation.md#unlinking-an-emptied-page`, and read them at open, which `load.rs` refuses today; test it with the header's capacity settable from tests.
- ⚠ **A single write or splice of 4 GiB or more corrupts the journal**: `Records::stash` keeps a record's length as a `u32`, so the record states a truncated length while the store counts all of it, and one of exactly `2^32` bytes even encodes the escape. `spec/journal.md#framing` bounds a transaction's length by `2^32 - 1`, so such a write must fail with `Error::OutOfBounds`; only the bound of `2^48` is checked today. No application writes values of 4 GiB in one piece, but the result is a file that does not open, not an error.
- **Arithmetic on values read from the file is not checked**: the entry decode adds deltas to page numbers, ids and ends, and the journal decode adds a move's `to_delta`, without checking for overflow, so a file whose CRCs match but whose numbers do not panics in a debug build and wraps in a release build, which can accept entries out of order that the delta encoding is meant to make unrepresentable. kladde-cpp checks each of them.
- **`FileStorage` takes no lock**, where `spec/file-format.md#concurrency` has a second process refused cleanly, and `create` truncates whatever file is there; kladde-cpp locks with `flock` before it truncates.
- **A `CutId` that escapes a transaction rolled back by `Store::atomically` can match a later record** that reuses its index with a cut of the same length, and a paste of it then pastes that record's bytes without an error. The typed layer never lets a cut out of a failed transaction, which poisons the store; only direct users of `Store` can meet it.

- **The cut never gives fillers to the header**: `impl/flush.md#the-cut` has the last leaf take them, or the header when there is no leaf, and kladde-rs fills the last leaf only.
- **The budget loop filters offers by their floors before it picks the best ratio**, where the pseudocode of `impl/consolidation.md#the-budget-loop` picks first and stops if the pick fails its floor; a table offer can then win when the data offer fails the fill floor.
- **The fold's output is not checked against the tracked length**, which `impl/write-phase-state.md` asks for as a cheap guard against a fold bug.
- **The decode recurses without a limit on its depth**, where `impl/opening.md` asks for a limit or an iterative decode, so a file that nests deeply enough overflows the stack.
- **A container's index panics on sizes of 4 GiB or more**, where `spec/span-table.md#bounds` asks every limit to fail cleanly: the Fenwick tree keeps sizes in `u32` cells, and converts with `expect`.
- **The load checks a reached `Ref` against its page's frame, not against the page's `content_size`.** It refuses a `Ref` into page 0 or 1, into a page that is not a `Data` page of the header's world, or reaching into the framing, but not one that reaches past the content into the padding.
- **Nothing is kept in the private chain**, where `impl/consolidator-state.md` designs what consolidation would carry there: every session starts its budget from `Options::budget_pages`, its rotating walk at the sequence's start rather than where `impl/opening.md` seeds it, and each span's heat from the epoch of the page that holds its entry.
- **A span costs about 90 bytes in memory, not 32.** The span table takes 48 bytes per span, since each record also holds the span's chain, its rope node, its heat, its entry's size and place in its table page, and its place in its data page's reverse list, which `impl/in-memory-state.md` keeps beside the span table; and the rope is an implicit treap of 40 bytes per node rather than a counted B-tree of about 15 per span.
- **`DurableVec`'s index takes 40 bytes even where it is empty or the elements are fixed-size**: a `Vec` of cells, its cached total, and the count's varint length. The total can live in the tree's allocation, and the count's length is the vector's to pass in, which would leave 24 bytes, or 8 with a thin vector.
- **None of the three refinements of `impl/consolidation.md` is implemented**: evacuation writes a victim's spans as they are, splitting the one that crosses the end of the page it fills, and merges no neighbours; the walk merges runs of cold spans and renumbers nothing; and the churn floor, budget controller and free filling are the address-table design's, unchanged, over pages of spans.
- **Compaction mode counts holes by scanning the page table once per flush.** `impl/consolidation.md#compaction-mode` promises `O(1)` amortised; kladde-rs keeps the cached index of the highest live page it describes, but counts the reusable pages below it in `O(pages)` at the start of every flush, which is small next to what a flush writes.
- **The budget loop never continues the last page of `pack`.** `impl/consolidation.md#the-budget-loop` lets that page cut a victim's span whenever the loop opens another page, so that only the flush's very last data page can close short; kladde-rs's loop always opens fresh pages, so `pack`'s last page is filled by free filling alone, and can close short even when the loop goes on.
- **What `rust/tutorial/durability.md` says about `close()` was measured on the address-table design, and has not been checked since**: that `close` truncates after the last live page only, so that a freshly created file keeps free pages below its data until a flush or two have consolidated them.
- **The header's eviction ranks by heat alone**, rather than evicting the coldest runs contiguous in id, and merges each evicted span with its neighbours in the sequence, extending each run while a neighbour whose entry also goes to a leaf makes a pair shorter than the bound.
- **A value is encoded into a buffer of its own, which the store copies into the record**, a copy that a store method taking an encoding closure would save.
- **Cuts exist on `DurableVec` only**: its `cut`, `insert_cut` and `push_cut`; `impl/values.md#moving-a-value-between-containers` has every method that takes a value take a cut too, but a plain `Vec`, the map, fields, `Option` and enums neither make nor take them.
- **`split_at_mut` and the count of changes are not implemented**, only `parts()`, for derived structs and enums and for tuples. `split_at_mut` was left out only because the evaluation needs it nowhere; the vector's Fenwick tree lies in cells and answers any element's offset, so it would serve as the two runs' index as it is.
- **`parts()` on a temporary `Mut` works in a `match` but not in a `let … else`**, which drops the temporary at the end of the statement. A consuming `into_parts(self)` would make the `let` form work too; it would need a line in `rust/derive-macro.md` and `rust/tutorial/deriving.md`, and is proposed rather than built.
- **`MemoryStorage` models a power cut as any subset of the writes since the last sync, each write**

whole. It does not model a write torn within a page; the page CRCs are meant to catch those, but no test exercises them yet.

- **Misusing the transaction and batch calls reports `Error::Corrupt`.** Ending a transaction that is not open is a caller's bug rather than a damaged file, and deserves an error variant of its own; the typed layers pair the calls through `Transaction`, `Batch` and `Region::transaction`, so only direct users of `Store` can hit it.

37.2 Choices kladde-docs leaves open

Where kladde-docs leaves a question open, or allows any answer, and what the implementation chose.

- **The journal is replayed by folding the recovered transactions in a flush at open**, before the decode, so that the decode reads the state after the last of them and needs no replay of its own; `impl/opening.md#replaying-through-a-flush` allows it.
- **The rope is an implicit treap**, one arena type of under 400 lines that also serves the fold's piece table and the journal's cuts, since a split at a position and a join of two trees are all three need.
- **The span id allocator hands out the id after the predecessor's, if it is free, and the lowest free id otherwise**, which keeps the spans a flush makes one after another consecutive without looking for a run.
- **The flush makes a run of 32 zeros or more among new bytes a `Zero` span.**
- **The span policy's constants**, chosen by the measurements of `evaluation/one-sequence.md`: `Options::span_min`, the bound L, 64 bytes; `Options::rewrite_below`, 64 bytes; `Options::inline_threshold`, 64 bytes; `Options::cold_flushes`, 16; `Options::walk`, 256 spans; and `Options::mu`, 0.02.
- **A `Box<T>`'s `FIXED_SIZE` is `None` whatever `T`'s**, since a type that holds itself through a box would otherwise ask its own size of itself while rustc evaluates it, which rustc refuses as a cycle (E0391); a vector of boxes then keeps an index it could do without.
- **The `Mut` of a `Fixed<T>` replaces the value whole**, since a `Mut` of one of `T`'s fields would have to write that field in its fixed form, which would take a second `Mut` for every type.
- **The map's hash table is its own**: open addressing over 32-bit positions, probed linearly and at most half full.
- **`FileStorage` grows the file with `set_len`, not with `fallocate`.** The new region stays unallocated until it is written, so a full disk shows up as a failed write or `fsync` during a flush, which poisons the store, rather than as a failed growth before the flush starts; `spec/durability.md#headroom` asks only that failure be clean, which poisoning provides.
- **The controller moves the budget multiplicatively, and measures fill against the whole file**: after each commit it divides the live bytes by the capacity of every page but the headers, and multiplies the budget by $\exp(4 \cdot (\tau - \mathrm{fill}))$, within `budget_min` and `budget_max`. Counting only live pages would let holes go unnoticed.
- **A page's record holds what its state has, and nothing else**: an enum over the page states, with a state of its own, filling, for pages a flush takes, and journal pages apart from live ones; only a live page has an epoch and a victim bucket.
- **The churn floor also bounds each victim**, and an offer takes whole victims by score, not by best fit.
- **Free filling moves part of at most one victim per page, and the victim stays eligible.** `impl/consolidation.md#packing-in-sequence-order-with-look-ahead` lists part of a victim too big to take whole as the last filler without saying how many victims to try; one per page is enough, and excluding a victim after a partial move would starve the budget loop of exactly the sparse pages it wants.
- **`Kladde::create_in` and `Kladde::open_in` take any `Storage` and `Options`**; `Kladde::create` and `Kladde::open` use a file and the default options, and `Kladde::new` is `create_in` with a `MemoryStorage`, which returns no `Result`, since a file in memory cannot fail to be created.

Chapter 38

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 balanced 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 kladde-rs and kladde-cpp).

38.1 The documents

document	contents
In-memory state	the structures a loaded file is held in: the span table, the rope, the page table, the reverse index, and the span id allocator
Opening a file	the steps from the headers to a decoded root value
Values and their positions	regions, what each container keeps to find its children, what a handle costs, handles that outlive edits elsewhere, the built-in containers, and moving values between containers
Write-phase state	what is tracked between flushes, and why it is derived from the journal
The flush	folding the journal into pages: the piece table, mapping pieces to spans, the span policy, placement, and the cut of the span table
Consolidation	reclaiming garbage: victim selection, evacuation, the page rewrite, merging spans, and the budget
Consolidator state	what consolidation could carry from one session to the next in the private chain
Transactions and batches	grouping operations into transactions, and the buffering state machine
Implementation notes	where kladde-rs departs from these documents, and what it found

38.2 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 span table's tree of child references is a *directory*, not a search structure: nothing ever descends it by key.
- The order of the content lives in the spans' links, which a reader walks once, at open; a tree over the spans is kept in memory only.
- 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(entries + n)` for a sequence of `n` bytes, and that the whole sequence is decoded into memory. What it buys is that every read is an in-memory read of the application's own values and every flush writes only deltas.

A future feature called *segments* would address the cost of bulk reads by letting application authors declare a subtree as a chain of its own, loaded lazily.

38.3 Overhead and work independent of size

The live pages exceed the sequence by three terms — the description, the garbage, and the page framing — and each can be held at a target that does not depend on the sequence's length, at a cost per modification that does not either; the file adds to them a number of pages that depends on what a flush writes, and not on the sequence's length either. This is what the design is for, and the algorithms on the following pages are chosen to keep it.

38.3.1 Where the bytes of the file go

Let `n` be the length of the sequence (with the schema's and the private chain's, which are small), `k` the number of spans the chains reach, `c` the mean encoded size of their entries without inline payloads, and $\bar{L}$ `= n / k` the mean length of a span. The live pages hold:

- every byte of the sequence once, in a data page or as an inline payload in a table page, and zeros nowhere: at most `n`;
- `k` entries: $k \cdot c = n \cdot c / \bar{L}$;
- garbage and slack, `G`: bytes of data pages that no reached span shows, dead entries in table pages, and room that a writer left empty;
- the page framing, 0.37 %.

So live pages come to about $n \cdot (1 + c / \bar{L} + G / n + 0.004)$.

The file holds more pages than the live ones:

- the **fallback** pages, which the last commit stopped referencing, and which the reuse rule keeps until the next flush's `fsync`;
- the **reusable** pages that the next flush will write into, freed by the commit before the last;
- the **journal**'s pages, at most its budget;
- **holes**: further reusable pages below the last live page, which a fall of the live size leaves behind, and which later flushes fill, or compaction mode and truncation give back.

38.3.2 Each term, and what keeping it costs

The description, $c / \bar{L}$, is set by the span policy. Bounded spans keep $\bar{L} \geq L / 2$ in the leaves, so $c / \bar{L} \leq 2c / L$: with entries of 8 bytes and `L = 256`, at most 6 %, and the header's exempt spans add at most a page. Each modification creates a constant number of span boundaries, at most two for a splice or an overwrite and three for a move, and removing one costs at most `L` bytes written again and one entry, so the policy costs `O(L)` bytes per modification, amortized. Merging cold regions into spans of a page lowers the term for them to about 0.15 %, once per region and cooling.

The garbage and slack, `G / n`, are set by the cleaner's target. Holding pages at a mean live fraction `u` leaves $G \leq (1 / u - 1) \cdot n$, 11 % at `u = 0.9`. In steady state, content that the application supersedes

turns into garbage, and the cleaner must win back as much room as that content took. A cleaner that takes the emptiest page first takes one at most `u` full, live fraction `v ≤ u`; writing its survivors into pages that it fills to at least `1 − θ` takes `v / (1 − θ)` of a page, so it wins back `1 − v / (1 − θ)` of one. Each page of room won back then costs `v (1 − θ) / (1 − θ − v)` pages of survivors moved, plus their spans restated, `c / L̄` per byte, and the cleaner can keep up only while `u < 1 − θ`:

`u`	`θ = 0`	`θ = 0.05`
0.85	5.7	8.1
0.9	9	17

That is the classic bound for log-structured cleaning (Rosenblum and Ousterhout, "The Design and Implementation of a Log-Structured File System", SOSP 1991), and uniformly random overwrites are its worst case; skewed workloads clean far more cheaply, which is what age terms in victim selection try to exploit. `θ` is a worst case too: packing holds every page a flush writes but its last to at least `1 − θ` full and usually fills it, so the cost lies near the column for `θ = 0`.

The pages around the live ones are set by what one flush writes. In steady state, a flush frees about as many pages as it writes, so with `p` pages written per flush, the file holds about `p` fallback pages and `p` reusable ones: `2p` pages besides the live ones, plus the journal's. `p` is the flush's new data and table pages, plus the survivors its cleaning moves: about ten pages for a flush that supersedes a page of content at `u = 0.9` and `θ = 0`. So a large file carries some 25 pages besides its live ones, about 100 KB, which is 0.1 % of a sequence of 100 MB and vanishes as `n` grows; for a file of a few pages, the same pages are most of the file.

Everything else is constant or logarithmic. A splice, an overwrite and a move state a constant number of spans; a moved byte restates its span and nothing else, since no key changes; the rope finds and edits spans in `O(log k)`; a handle is made from its parent's with one lookup in the parent's index, and a change of size updates one entry per level (what a handle costs); victims come from bucket queues in `O(1)`, and their spans from a reverse index in `O(1)` each; and the table pages' directory changes `O(depth)` pages per leaf added or removed, amortized over the entries a leaf holds.

So the live pages' overhead is `c / L̄ + G / n + 0.004`, set by `L` and `u`, the file adds `2p` pages and the journal's, and the work per modification is `O(w + L)` bytes for `w` bytes written and `O(log n)` time in the store, plus the walk from the root to the modified value, which the application makes anyway.

38.3.3 A worked example

Arithmetic, not a measurement: a sequence of 100 MB, of which 1 MB is being edited.

part	spans	entry bytes	share of `n`
99 MB cold, in spans of a page	24,300	146 KB, at 6 bytes each	0.15 %
1 MB hot, in spans of 128 bytes on average	8,200	82 KB, at 10 bytes each	0.08 %
garbage and slack, pages kept 90 % live			11 %
page framing			0.4 %
live pages			**about 112 %**
fallback and reusable pages, a flush writing 12 pages, and a journal of 4			0.1 %
the file			**about 112 %**

At `u = 0.85` the garbage takes 18 %, and the cleaner moves at most 5.7 bytes per byte written instead of 9. In either case, it is the cleaner's target, and not the description, that decides where between 10 and 20 % the file lands. The file as one sequence, measured shows what kladde-rs reaches on real edits.

38.4 The mirror

An implementation reads pages into its own buffers with ordinary file reads, rather than mapping the file, and reads only the pages the chains reach. At open it reads every table page the header reaches, walks the chains, and then reads the data pages the reached spans show, in page order, into a **mirror**, from which the decode of the sequence reads; Opening a file gives the steps.

It might be tempting to load each data page lazily, the first time something reads from it. Reading them eagerly turns scattered reads into one sequential pass over a set known in advance, and validates every reached page's CRC at open rather than at first touch — which may be long after the last backup that could have helped. And it keeps the resident set a property of the *file*: nothing obliges a data type to read anything, and a tool that only compacts reads no content at all.

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 span's bytes in a data page are read again only when a flush moves them, merges them with their neighbours, or rewrites a short span an edit cut into — all rare, and all able to afford a fresh read.

Peak memory comes at open, while the mirror and the decoded values coexist: about the sequence's size twice over, until the decode ends and the mirror goes. Holding the mirror for good instead, and reading values from it rather than decoding them into the host language's vectors and strings, would halve that peak and save every in-memory copy, but it is an open idea: values would no longer be the host language's own, reads would need the store, and the journal's edits would have to reach the mirror too.

Table pages stay resident by default, for a reason data pages do not share: `Inline` payloads live inside them, and every flush that restates an inline span copies its payload. They are also small next to the sequence. Nothing depends on it, though: only the header must stay, being the write buffer, and dropping the leaves after load costs one read per leaf a flush rewrites or restates from.

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. Should that trade ever be revisited, 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. The decode reads the sequence through a cursor that follows the rope across span boundaries and hands out slices of the mirror and of the table pages, so no byte of the sequence is copied before a value's own decoder takes it.

Chapter 39

In-memory state

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

The store keeps a span table, a rope, the page table, a reverse index, and a span id allocator. Together they take about 32 bytes per span, besides the resident table pages, and none of them grows with the number of values the application holds: positions of values are kept by the values' containers, Values and their positions says how.

Integer widths below follow from the bounds the specification states, and are given because they are the same in any language.

39.1 1. The span table

An array indexed by span id, one record per span a chain reaches:

```
struct Span {
    content: Content,     // what the span shows
    len: u32,             // its length in bytes, never 0
    next: u32,            // the span after it in its chain, or 0
    entry_page: u32,      // the table page that holds its live entry
}

enum Content {
    Ref { page: u32, offset: u16 },   // bytes of a data page
    Inline { offset: u16 },           // its payload, inside its entry in `entry_page`
    Zero,                             // zeros
}
```

That is about 16 bytes per span. **An array, because span ids are dense integers** that the allocator hands out lowest first, so every lookup is an index rather than a hash. An id no chain reaches has an empty record, and is free.

The in-memory span table is the on-file span table resolved: it holds, for every reached span, what its newest entry says, and nothing about older entries. Content is never copied out of the pages while they are resident: a `Ref` names its bytes in a data page, and an `Inline` names its payload inside its own entry, which lies in a resident table page.

39.2 2. The rope

The rope is a counted balanced tree over the sequence's spans in order, each item a span id and its length, each node the total length below it. It finds the span that holds a position, splits one, and cuts and inserts runs of spans, each in `O(log k)` for `k` spans, which is what the flush needs to map the journal's positions to spans, and what the decode follows to read the sequence in order.

A counted B-tree with leaves of 32 entries is the recommended shape: about 11 bytes per span in its leaves at a typical fill, a few percent more in interior nodes, and 4 bytes per span in the span table for the leaf that holds it, so about 15 bytes in all; and a lookup descends three levels for any realistic file, each a scan of a node of one or two cache lines. An implicit treap, a randomized binary tree whose nodes know their parent and the total length below them, does every operation with two primitives, a split at a position and a join of two trees, and those are exactly what the fold needs as well, so one arena type can serve the rope, the fold's piece table and a transaction's cuts. It costs a node of about 40 bytes per span, and `O(log k)` expected steps, about `2 ln k - 3` levels on average, each likely a cache miss; that shows only once memory per span matters, 640 MB against the B-tree's 240 for 16 million spans.

The schema's and the private state's chains are short and written only by flushes, so a list of span ids serves them.

39.3 3. The page table

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

Coverage counts what the page holds that a reached span relies on: for a data page, the bytes reached spans show in it; for a table page, the bytes of the live entries in it, inline payloads included, and of its child references. It is a 32-bit counter rather than a page-sized one, because a page's live bytes exceed its capacity when several spans show the same bytes, which is legal and bounded precisely so that 32 bits suffice. A page whose coverage reaches zero is reusable once the next commits have passed it through the reuse rule's quarantine — a table page only once it is also unlinked from its parent (unlinking).

A page's record holds what its state has, and nothing else: an enum over the states of the specification. A page a flush takes is *filling*, which the specification counts as reusable: it has a kind and coverage but no epoch, since its epoch is the flush's, and no place among the victims. The commit turns every page it wrote into a live one. Journal pages are a state of their own, although the specification counts them live, since the journal keeps what they hold.

Live pages are bucketed by live fraction — a handful of buckets suffices — which 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 `Table` pages are bucketed separately, and header pages in neither, since a header is rewritten every flush and so is never a victim.

An ordered set tracks reusable pages, handing out the lowest first, which keeps the file's live pages low, where truncation can follow (where the pages come from).

39.4 4. The reverse index

For each data page, the reverse index lists the spans that show bytes in it, so that evacuation finds what a victim holds in `O(1)` per span. A data page is opaque bytes with no ids in it, so decoding it tells nothing; but span ids are stable, and a span's address changes only when the span is stated with a new one, so the list is exact and cheap to keep: a span is added to its page's list when it is stated with a `Ref` there, and removed when it is restated elsewhere, merged away, or no longer reached. Keeping each span's place in its page's list in its span record makes both `O(1)`.

39.5 5. The span id allocator

The allocator hands out the lowest free id, and runs of consecutive free ids where a flush states consecutive spans, from a bitmap of free ids with a summary bit per word, which finds either in a few scans of words. Low, consecutive ids are what keep id deltas short and successors implicit in the entry encoding. A free list threaded through the span table's empty records would cost no memory of its own, but would hand ids out in the order they were freed, which scatters them.

An id is free the moment its span is no longer reached, and may be handed out again in the same flush: the reach rule has the new span stated in the epoch it becomes reached, so its entry is newer than any old entry of the same id.

39.6 Liveness is one bit

An entry is live exactly while it is the span table's entry for a span the rope, or a short chain, holds, so liveness needs no counter at all: no pins, no count of mentions, no size or tombstone to keep alive. A dead entry survives as bytes in its page until a rewrite of that page drops it, and the page's coverage stopped counting it the moment it died.

39.7 Heat

Each span carries its heat: the epoch of the flush that last wrote its bytes, or the number of flushes since. The flush ranks the header's entries by it (the header as write buffer), and the rotating walk merges only spans that have gone cold (merging spans). It is content age, where a page's epoch is location age: restating a span re-stamps its entry's epoch, but not its heat.

39.8 During a flush

A flush adds transient state of its own, all dropped when it commits:

- **the fold's piece table**, over the sequence as the journal leaves it (the fold);
- **the new bytes**, of new spans and of runs that have no span yet, in a scratch buffer, each with whether its entry will hold them inline;
- **the stated set**: the spans that get a new entry in this flush, and which of them go to the header.

Coverage releases happen as the flush changes spans, so victim selection sees the state the flush will leave; charges wait for locations: at placement for bytes, and at the cut for entries. A flush that fails must therefore unwind or poison the session.

39.9 Epochs belong to pages, not to entries

An entry inherits the epoch of the page it sits in, so restating an entry re-stamps it. The header is rewritten every flush, so every entry in the header carries the current epoch, and two epochs can disagree about a span only between the header and a leaf, or between two leaves.

39.10 Memory for very large files

The span table and the rope take about 32 bytes per span together. A cold gigabyte in spans of a page needs about 8 MB of them; a gigabyte edited everywhere at once, in short spans, would need far more, which is an open question, and what segments would bound.

Chapter 40

Opening a file

The steps from the two headers to a decoded root value, and what each checks.

Opening reads every live table page once, walks each chain once, reads the data pages the chains show in page order, decodes the sequence, and replays the journal: `O(entries + n)` for a sequence of `n` bytes. There is no sweep resolving bytes across epochs, no clipping at sizes, and nothing to resolve per byte.

40.1 The steps

1. **Read both headers and take the governing one**: the CRC-valid header with the highest epoch (recovery). `fsync` the file once, which makes that header durable and retires the older world at once.
2. **Read every table page the header reaches**, through its child references, checking that they form a tree, and keep, for each span id, the entry of the highest epoch in the span table. No sorting is needed: an id's entries are compared by their pages' epochs as they come. Each page's framing is checked, and its epoch must not exceed the header's.
3. **Walk each chain from its head**, building the rope for the sequence and lists for the short chains. Refuse the file on an id no entry describes, a span reached twice, or a span two chains share; count each reached span's entry, and its bytes, toward its pages' coverage, and enter it in its data page's reverse list. Entries of spans no chain reaches are dead, and their ids free.
4. **Read the data pages the reached spans show, in the order of their page numbers, into the mirror.** Each must be a `Data` page whose framing checks and whose epoch does not exceed the header's, and each `Ref` must lie inside its content. Reading them in the sequence's order instead, as step 5 needs them, would jump between pages wherever the sequence was ever edited; step 3 has already found which pages are needed, so step 4 reads them in one sequential pass.
5. **Decode the sequence through a cursor that follows the rope**: the root type's decoder reads front to back, and every container builds its index of children as it goes. A recursive type nests as deeply as the file says, so the decode limits its depth, or does not recurse, and refuses a file beyond the limit rather than overflow its stack. It refuses an encoding that is not canonical, and bytes left after the root value. Then drop the mirror.
6. **Replay the journal**, applying each recovered transaction's records to the in-memory values as well as to the store's structures.

If the header's root fingerprint differs from the application's root type's, step 5 decodes by the writer's schema chain instead, resolving it once against the application's types, or refuses the file if resolution is not implemented.

40.2 Replaying through a flush

An implementation may fold the recovered journal in a flush at open, before step 5, so that the decode reads the state after the last transaction and needs no replay of its own. It has to fold a non-empty

recovered journal before appending to it anyway (the start of a session), and replaying records into in-memory values would need every container to accept byte-level records it did not make. The price is that a file whose journal was not empty costs a flush to open.

Recovery reads the payload pages the journal names, to check each reference, and keeps their bytes in the buffer with where they lie, so that the flush at open takes those of its own epoch into the file as data pages, as any flush does (payload pages); those of a later journal, reached through a seal, it writes again.

40.3 The seeds of the rotating walk and of heat

What the file does not record, open seeds:

- **the heat** of each span, from its entry's page epoch, which can only understate how long its bytes have gone unwritten — content is never younger than the page that holds its entry — or from the consolidator state, if one is current;
- **the rotating walk's position**, from the consolidator state, or else at a span chosen by the governing header's CRC modulo the number of spans: deterministic for a given file, so tests reproduce, and spread over the sequence across sessions.

Chapter 41

Values and their positions

How the values an application holds find their bytes in the sequence: regions, what each container keeps to find its children, what that costs per mutation, handles that live long, the built-in containers, and moving a value from one container to another.

No value stores where it lies, and neither does the store: the store knows bytes, each container knows where its children start relative to itself, and the chain of handles from the root composes the two for as long as it lives. A *handle* is what a value is mutated through, `Mut` in kladde-rs and kladde-cpp; a binding in another language may intercept mutation differently, but it needs the same positions, and the algorithms below are the same.

41.1 A value is a region

A data type sees only its own region of the sequence: where its value starts, resolved when its handle is made, and how long it is. It writes and splices at offsets relative to that start, checked against its length, and it neither knows where the region lies nor can reach anything outside it. A region offers:

- `write(at, bytes)`, which overwrites bytes in place;
- `splice(at, removed, bytes)`, which replaces bytes and shifts what follows; it records a `Write` wherever `removed` equals the number of bytes inserted, so a whole-value `set` calls it whether or not the size changes;
- `splice_value(at, removed, value)`, which encodes the value straight into the record, asking it for its encoded length once, since the record states the length before the bytes, and returns that length to its caller;
- `move_bytes(from, len, to)`, which reorders without writing;
- `paste(at, removed, cut)`, which inserts a cut (below);
- `child(at, len, index)`, which makes the region of one of the value's children, together with a note that it is child number `index`, so that when the child changes size its region tells the value's index of its children.

`write` and `splice` return the cut of what they remove. Every offset is relative to the region's start, and every method refuses one outside it.

The obligations of a data type are four: its fixed size and encoded length match what its encoder writes; its encoder and decoder are inverses; every mutation leaves its region holding the value's encoding when its transaction ends; and its descriptor matches the bytes. The third is checkable, which an ordering discipline over allocations was only by replay: a store in a test can keep the sequence in memory and compare a region with the value's encoding after every method.

A push onto a vector of variable-size elements, for instance, splices the element in at the end and rewrites the count, in one transaction, and only then changes memory:

```
fn push(&mut self, value: T) -> Result<(), Error> {
    let n = self.vec.items.len();
    let len = self.region.transaction(|r| {
```

```
        let len = r.splice_value(r.len(), 0, &value)?;
        r.splice(0, varint_len(n), &varint(n + 1))?;
        Ok(len)
    })?;
    self.vec.sizes.push(len);     // appends to the Fenwick tree, O(log m); nothing for fixed-size `T`
    self.vec.items.push(value);
    Ok(())
}
```

Recording first and changing memory after means that a failed append leaves the value as it was. In a language whose allocations can fail without ending the process, such as C++, a mutation also does everything in memory that can fail before it records — reserving capacity, constructing the new element — so that what follows the record cannot fail.

41.2 Positions belong to containers

A container hands each child a subregion and keeps, in memory, where each child starts relative to itself: by multiplication for fixed-size children, by an index otherwise. A child that changes size reports the change to its container, which updates its index and reports its own change to its container in turn, up to the root.

- **A struct keeps no index at all.** Its fields up to its first variable-size field start at constants; after that, each starts at the sum of the encoded sizes of the fields before it, a number of additions fixed by the struct's type.
- **A container of fixed-size elements keeps no index either**: element `i` starts at `i` times the element's size, after the count.
- **A container of variable-size elements keeps an index**, chosen below.
- **Every container caches its own encoded size**, so that adding up sizes never recurses.

A fixed-size type buys three things, all in memory: offsets inside it that the compiler can fold, containers of it without an index, and writes that never change its size, so nothing above it has to hear about them. The choice is the application author's, per field and per element type: varints and unpadded enums save bytes on file, and `Fixed` saves memory and time in containers that hold many values, and keeps a field that changes often, such as a counter, from changing size whenever it crosses a power of 128.

Why not a global index in the store. One index of anchors, one for every value whose position anyone may ask for, would cost a node of an order-statistics tree per value, where a container's index costs four bytes per variable-size element and nothing per fixed-size one, and it would teach the store about values, which nothing else requires it to know.

41.3 What a container keeps to find its children

A container of variable-size elements has five ways to keep where they start, and memory decides between them.

index	finding an element	a size change	an insertion	memory per element
an array of starts	`O(1)`	`O(m)`	`O(m)`	4 bytes
a Fenwick tree of sizes	`O(log m)`	`O(log m)`	`O(m)`	4 bytes
a counted B-tree	`O(log m)`	`O(log m)`	`O(log m)`	about 8 bytes
a tree weighted by the elements' sizes	`O(1 + log(W / w))`, amortized	the same	the same	about 16 bytes
a start every 64 elements	`O(64)`	`O(m / 64)`	`O(m)`	1/16 byte

`m` is the number of elements, `W` the container's encoded size and `w` the element's, and memory assumes sizes recorded in 32-bit values; a container that would outgrow them fails cleanly.

A Fenwick tree is the default: a size change of an element, typing into a string inside it, say, is the frequent case and costs `O(m)` in an array of starts, while an insertion costs `O(m)` in the array that holds the elements anyway. The runtimes here are for lookups that happen a handful of times per mutation, when the mutation happens and not during a flush, so they matter less than memory: kladde is meant for interactive applications whose mutations follow user actions, and a Fenwick tree's lookup or update over a million elements is about twenty steps, well under a microsecond. For bulk operations, such as pasting large content from the clipboard, an application should build the value outside any handle and bring it in whole, which encodes it into one buffer and splices it in.

A Fenwick tree over blocks of 32 elements would keep one sum per block, an eighth of a byte per element, and find where element `i` starts from the tree's sum up to `i`'s block plus the encoded sizes of the up to 31 elements before it in the block, which it asks of the elements themselves; for small elements such as varints that scan costs about what the tree's own steps cost. It is the better choice for vectors of small variable-size elements, where a full tree would double the memory.

Open: whether a Fenwick tree is the right default, what an application with millions of variable-size elements should use, and whether any data model nests deeply enough for size-weighted indexes to pay.

41.4 What a handle costs

Making a handle costs one lookup in its parent's index, however deep its value lies, because the parent's handle already holds where the parent starts. Every handle holds the start of its value, which it needs anyway, since journal records name positions in the sequence. An application reaches a value one level at a time — `drawing.get_mut().shapes_mut().get_mut(7)?` — and each step adds its child's start relative to the parent to the parent's start:

- a field before its struct's first variable-size field, or an element of a container of fixed-size elements, starts at a constant, or at `i` times the element's size;
- a field after it starts at the sum of the encoded sizes of the fields before it;
- an element of a container of variable-size elements starts where the container's index says.

The start a handle holds stays right for as long as the handle lives: the handle borrows its parent exclusively, so nothing before its value can change size meanwhile, and its value's own changes of size move only what lies behind it. A language that cannot enforce the exclusive borrow makes it a rule, the kind iterator invalidation is: a handle is valid until something is written through a handle that is not its descendant; a debug build can check it with a stack of the live handles that a write pops down to the handle it went through. Iterating needs no index at all: a handle that steps through a container's elements adds each element's size to the start of the one before.

A change of size walks the same chain back up, since it changes the encoded size of every value around the one that changed: each ancestor updates its cached size and its index's entry for the child the change came from, one update per level.

So the depth is not a cost of the design, and the logarithm per level is a choice of index. The costs fall at three different times: making a handle costs a lookup at each level it descends, once; a write through it that keeps its value's size costs nothing more; and a write that changes its value's size costs an update at each level above the value. The chain from the root is walked to make the handles anyway, so a constant per level adds nothing that grows with the file.

41.4.1 Handles made together

Handles made together — for every field of a struct or of an enum's current variant at once, or for two runs of a container's elements — ask for their start when they write instead of holding it, since a sibling before a handle can change size while the handle lives. Such a handle holds a link to its parent's index and its own number there, and computes its start whenever it writes, from the parent's start and the sizes the index keeps for the children before it. The index keeps those sizes in cells that a sibling's report of a change of size can update while the other handles still hold shared references to it. A write

to an earlier part that changes its size, followed by a write to a later part, then works out by itself: the first write's report updates the parent's index, and the second write computes its start from the updated index. Such a handle pays a lookup at every write rather than once; a count of changes kept in the index, compared with the count the handle saw last, would let it reuse its start until a sibling's size has changed. A field whose predecessors are all fixed-size has a constant start even then, since no sibling's change of size can move it.

A handle made on its own below a handle made together holds its start relative to that handle, since that handle's own start can move while it lives. Between the two, every handle borrows the one above it exclusively, so nothing between them can change size before it, and the relative start stays right; above them, a sibling's write can shift everything. A write then computes its absolute start by climbing only the handles made together: at each, one lookup in the index of the parent it was made from, plus the relative starts of the handles made on their own in between, which are constants. A write that changes its value's size updates every level up to the root, the handles in between included, so every level keeps its cached size and its index in cells.

The test for this: in a tree, take two sibling handles of the root together, descend two levels in each on their own, take two siblings of each of those together, descend two more levels in each of the four; then write in the four remote cousins, changing their sizes, and check that each writes at the right position after the others have changed size, that the indexes at every level match the in-memory values, and that the file then holds the value's encoding and reopens to it.

41.4.2 Indexes weighted by size

Indexes that weigh elements by their encoded size make the whole chain cost `O(depth + log n)`, as one tree over every value's position would, with no such tree. In a tree over a container's elements in which an element of `w` bytes, in a container of `W`, lies at depth `O(1 + log(W / w))` — a biased search tree (Bent, Sleator and Tarjan, "Biased Search Trees", SIAM Journal on Computing, 1985), or a splay tree, whose amortized cost of access is the same for any weights (Sleator and Tarjan, "Self-Adjusting Binary Search Trees", Journal of the ACM, 1985) — the lookups from the root, of `n` bytes, down to a value of `w` bytes cost `O(depth + log(n / w))` in all, since each level's `log(W / w)` cancels against the next level's. Changes of size telescope the same way on the way back up. That pays for deep nesting of large containers, a document tree whose nodes hold long lists of nodes, say; for shallow nesting a Fenwick tree per container is simpler, and the default.

41.5 Handles that outlive edits elsewhere

An application that keeps references into its document for as long as it likes needs handles that survive edits elsewhere, and a list whose elements know where they stand provides them. A document editor holds references to its nodes everywhere — in the selection, in tools, in dialogs, in the undo log — and writes through any of them at any time; a handle whose start goes stale at the next write elsewhere does not serve it. A language with a borrow checker can forbid holding such references, at the price of an API that the application must be designed around; every other language — C++, Python, Java — needs these handles for applications that were not.

A list keeps its elements at stable addresses, each element knows its parent and its own index among its siblings, and the list keeps a Fenwick tree of its elements' sizes. A reference to an element is then valid until the element is removed, and computes its start when it writes by climbing from the element to the root: at each level, the element's offset among its siblings, from the Fenwick tree at the element's index, plus the offset of the list in its parent. That is `O(depth · log m)` per write, the cost of handles made together at every level, and a change of size updates the same path back up.

- **An insertion or a removal renumbers the elements behind it and rebuilds the Fenwick tree**, `O(m)`, which is what moving the elements' pointers in the list's array costs anyway; finding an element's index is a load.
- **An element costs a parent link, an index and a cell of the tree**, about 16 bytes, which a document tree whose elements each hold a name, attributes and children affords.
- **An order-statistics tree with parent links** would let an element find its index by climbing, and insert in `O(log m)`, at 24 to 40 bytes per element; it pays only for lists of hundreds of thousands of elements edited in the middle, and can be an option for them.

So a vector and a list differ the way an array and a linked list do in the host language — compact, or stable — and an application chooses per container.

41.6 The built-in containers

Their encodings are the specification's Sequence, Map and Text; what they keep in memory is theirs.

- **A vector**: the count, then the elements; in memory, the elements in an array, and for variable-size elements a Fenwick tree of their sizes. A push and an insertion are a splice and a count, in one transaction; a removal likewise; a reordering a `Move`, which writes no element's bytes; a size change of an element a report that updates the tree, `O(log m)`. A vector of fixed-size elements needs no tree, so where the host language's own vector is durable, it serves for them.
- **A list**: encoded as a vector is, with elements at stable addresses, above.
- **A string**: UTF-8, then `0xFF`; in memory, the host language's string, or a rope for long text, which then needs no container of its own, since the file already takes splices in long text at constant cost. An edit of text is one splice, since text has no count to rewrite. Setting a whole new text compares it with the old one and splices only the range between their common prefix and their common suffix, as the specification guarantees.
- **A map**: the count, then the entries, each its key and its value; in memory, an *index map* — the entries in an array, in their order on file, and a hash table of their indices, so that each key is stored once — and for variable-size entries a Fenwick tree of their sizes. Removal swaps: the last entry moves into the removed one's place, by a `Move` that writes no bytes, the removed entry is spliced out, and the count goes down, so the file stays dense, with no tombstones, and the only index that changes in memory is the moved entry's. An insertion appends, and a lookup is a hash and an index.
- **An opaque payload**, such as a value serialized by a foreign library: its bytes, prefixed with their length. Setting a new payload splices the range between the common prefix and suffix of the old and the new bytes, and rewrites the length where it changes, in one transaction; where the changed range lies near the start, it writes the length and the range as one splice.

Removing a value is one splice, whatever it held: there is nothing to free, and nothing can leak.

41.7 Moving a value between containers

Moving a value within one container is one `Move` record and three to six span entries, whatever its size; between containers, cutting the value out and pasting it in costs the same, as long as both happen in one journal. Every value is its bytes, so restacking a group, reordering a list, or sorting it are `Move`s inside their container, and moving a subtree to another parent is the question. Five answers were weighed, and cut and paste within a journal was adopted:

- **Encode it again**: remove the value, and insert its encoding where it goes. Simple, and the way to insert a value more than once; but the journal and the flush carry all of its bytes. It stays as the fallback wherever a cut has ended.
- **A move at a common ancestor**: a method on the container that holds both places. A binding that forbids holding two handles at once, as Rust's borrows do, would have the ancestor take both paths, which is awkward to type and to derive.
- **Cut and paste within a transaction**: the bytes a record removes stay available until the transaction ends. It serves a move within one operation, such as dropping a dragged shape into another group, but not an application that removes a value in one transaction and inserts it again in a later one, as undoing a deletion does.
- **Cut and paste across flushes, through a chain of cuts**: the bytes of removed values kept reachable from the header until they are pasted or dropped. Deleted content would then stay in the file until the application lets go of the values, where a user deleting it expects the file to shrink; bounding the chain would need the store to evict cuts, after which a paste must fall back to encoding anyway.
- **Cut and paste within a journal**, which is adopted: the bytes a record removes stay available as a cut until the journal that holds the record is folded, and a later `Paste` in the same journal inserts them elsewhere, naming them instead of carrying them. Nothing goes to the file that would not be there anyway, so deleted content becomes garbage at the next flush, as after any removal.

A cut is a weak promise: pasting it is cheap while its journal lasts, and encodes the value again otherwise. A container's `cut(i)` removes element `i` as `remove(i)` does, and returns a cut — the in-memory value, the cut's number among the cuts of the journal, and the journal it belongs to. Methods that take a value — a push, an insertion, a field's `set` — take a cut too, and move the value into memory. A cut gives no mutable access to its value, since its bytes must stay the value's encoding until they are pasted, and it can be pasted at most once. Dropping it unpasted does nothing more: the value was removed when it was cut. Any method that splices a value out can return it as a cut: a field's replacement, an option's `take`, or a change of an enum's variant, for the payload of the old variant. And an application with references that outlive edits, above, can let a removed element remember its cut, so that inserting the same element again pastes it.

Whether a paste is a `Paste` or a `Splice` is decided when its transaction is appended, not when the application calls it. A flush can land between a cut and its paste — triggered by the paste's own transaction, which is preceded by a flush when it does not fit in what is left of the journal's budget — and folding the cut's journal ends the cut. So the buffer holds the operation as "paste cut `c`, or else these bytes", with the value's encoding available, and the transaction's size is computed with the `Paste`; if the transaction fits, it is appended with the `Paste`, into the journal that made the cut, and if a flush must come first, the paste is encoded as a `Splice` of the value's bytes, the size is computed again, and the flush triggers continue from there. Two cases need no fallback at all: a cut and its paste in one transaction, which no flush can separate; and the ready run of a batch, which is appended before the flush that cuts the batch.

What it costs: an implementation keeps what it needs to resolve the journal's cuts until the journal is folded — for the fold, the trees of pieces that the cuts removed — and the length of each cut, to check pastes before they are recorded. How long a cut lives is how long a journal lives, so an application that flushes often, when idle say, ends its cuts sooner; an undo right after a deletion pastes, and one after the next flush encodes again. Changing a value between its cut and its paste, say to adjust a shape's coordinates to its new group, would need records whose positions lie inside a cut rather than in the sequence; it is left out, since pasting first and changing the value then does the same.

Cap'n Proto has the same idea within one message: an *orphan* is an object disowned by one parent and adopted by another without copying, move-only, and zeroed if nothing adopts it.

Chapter 42

Write-phase state

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

42.1 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. The alternative — a snapshot of the changed content kept beside the log — would have to agree with the log about the order of every change, and the sequence's positions make order decisive: a splice at position 10 followed by a write at position 20 does something else than the two in the other order, since the splice moves what the write names. So the flush folds the log, and nothing else, into the new sequence.

42.2 Two derived structures

The sequence's length, as every record recorded so far leaves it, so that bounds checks during the write phase are `O(1)`: a record whose range does not lie inside the sequence is refused before it is recorded, and records nothing. It also checks the fold: the fold's sequence must come out exactly this long, a consistency check that costs one comparison per flush.

The cuts of the journal: for each cut, oldest first, its length, whether it was pasted, and which journal it belongs to — the epoch of the journal its record is appended to, or will be — so that a paste can be checked before it is recorded: it must name a cut that is not pasted yet, and its length gives the paste's. A flush drops the cuts that the records it folded made; cuts that records past the committed cursor made stay, since those records go to the next journal.

The pastes that may fall back: a paste recorded in one transaction names a cut that an earlier transaction made, and it is a `Paste` only if it is appended to the journal that made the cut. So the buffer keeps, beside each such paste, what its `Splice` would hold — the value's encoding, or the value to encode it from — until its transaction is appended, when the flush triggers decide which of the two it becomes (cuts).

The fold's output — the new sequence as runs of committed bytes and new bytes — is built at flush from the log alone, consumed by the mapping onto spans, and dropped; see the fold.

42.3 After a flush, and after a rollback

An explicit flush can run while a transaction is still open, with its records buffered past the committed cursor, and an automatic one holds back the transaction that did not fit (flush triggers). The length covers those records too, so it is right after the flush as it was before; the cuts those records made belong to the next journal, and stay.

A transaction that fails halfway and is rolled back moves the buffer's end back to where it began, and with it the length and the list of cuts, to what they were at its start.

Chapter 43

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

A flush folds the journal into the new sequence, maps it onto spans, lets the span policy merge what it wants merged, places new bytes in data pages with consolidation's victims riding along, cuts the span table into the header and leaves, and commits.

43.1 The fold

The fold keeps a piece table over the sequence, starting as one piece, the committed sequence, and the journal's records edit it. A piece is a run of the new sequence and where its bytes come from: the committed sequence, from some position on, as it stood after the last flush, or the journal's byte arena, from some position on.

```
enum Source {
    Committed(u64),   // the committed sequence, from this position on
    Arena(u64),       // the journal buffer's arena, from this position on
}

struct Piece { src: Source, len: u64 }
```

The pieces sit in a counted tree, as the rope does over spans, so each record costs `O(log p)` for the `p` pieces the journal made, however long the sequence:

record	effect on the piece table
`Write(pos, bytes)`	take `[pos, pos + len)` out, as a cut; insert one `Arena` piece
`Splice(pos, removed, bytes)`	take `[pos, pos + removed)` out, as a cut if it is not empty; insert one `Arena` piece
`Move(from, len, to)`	take `[from, from + len)` out and insert it at `to`
`Paste(pos, removed, cut)`	take `[pos, pos + removed)` out and drop it; insert the cut's tree

Taking a range out is two splits and a join, and returns the range as a tree of its own; inserting is a split and two joins. **A journal's cuts are kept as those trees until the fold ends**, so a paste inserts the pieces its cut removed and the flush states three to six spans for the two, however large the value, and writes none of its bytes, whichever transactions of the journal the cut and the paste lie in; replay at open does the same.

At the end, the fold lists the new sequence as pieces, merging adjacent pieces that continue each other. A record whose range does not lie inside the sequence, or a paste of a cut its journal did not make or already pasted, makes the journal invalid.

What falls out unasked:

- **A splice that lands in bytes the same flush wrote costs nothing beyond them**: it edits `Arena` pieces, and the bytes it shifts are never mentioned.
- **A move costs no bytes**: its pieces are committed ranges, which keep the spans that show them.
- **Repeated writes to one place cost one write**: only the last one's bytes survive in the pieces.

43.2 Mapping pieces to spans

A committed range is shown by the spans that showed it, found through the rope; where a range begins or ends inside a span, the span is split: its left part is stated again with a shorter length, and its right part becomes a new span over the same bytes. New bytes become new spans, inline below the threshold and placed in data pages above it, unless a payload page holds them already. Every span is linked to the one that now follows it, and every span whose content, length or successor changed is stated, with every new one.

So a splice cuts the chain at its two ends and adds the span of its new bytes: one to three entries. A move cuts it at three places — the start and the end of the moved run, and its destination — and splits at most one span at each; the three left parts are also exactly the spans whose successors change: the one before the run now links to what followed it, the run's last span to what follows the destination, and the span before the destination to the run's first span. Where a place is the sequence's start, the header's first span changes instead, which costs no entry, and where a place already falls between two spans, its split goes; so a move states three to six spans, however many it moves. Every span inside a moved run keeps its entry, since neither its bytes nor its successor change.

Spans that no chain reaches any more are dropped: their ids are free, and their entries dead. The reach rule holds by construction, since a span becomes reached only as a new span, or as a span whose entry the flush writes anyway.

43.2.1 Payload pages

Bytes that a payload page of the flush's epoch holds become a span over the page, and the page a data page of the header the flush commits, without a byte written. The journal's buffer records, beside its arena, which runs of the arena also lie in a payload page, and where (placing them); so the fold needs no source of its own for them, and its `Arena` pieces are mapped as before, except that each part of one that a placed run covers becomes a `Ref` over the payload page. The page is taken into the flush as a page it fills, so that the commit makes it live with the pages it writes, its coverage the bytes its spans show; a payload page that no span shows retires with the journal's pages.

- **Only payload pages of the flush's own epoch**: a flush that folds a recovered journal and, through its seal, the next one finds the next one's pages of a higher epoch than its own, which the epoch assertion keeps out of its header's world, so their bytes are written again, as new bytes.
- **Not a part shorter than `rewrite_below`**, which a later record of the journal cut out of a run: it is written again with the bytes around it, rather than keeping a page alive for a few bytes.
- **The page keeps its slack**, the room a payload page's last run left, until consolidation evacuates it, since the flush writes no page it takes this way.

43.2.2 Overwrites: split or rewrite

An edit that cuts into a span can split it, or write it again whole with the edit's bytes, and the flush rewrites a `Ref` shorter than a threshold and splits a longer one.

- **Splitting** costs three entries the first time a place is written, and one for every later overwrite of the same bytes, which restates the middle span; it leaves the description fragmented until the span policy merges the pieces.
- **Rewriting** costs one entry and the span's length in data bytes, and fragments nothing; it suits short spans, and is costly for long ones.

So a drag that moves a point many times splits its span once and restates an inline span of a few bytes at every later flush. **An `Inline` span is always rewritten**, since splitting one restates its whole payload as well, in two or three entries instead of one. The threshold applies to the span an edit cuts into, not to the parts a cut leaves: a `Ref` long enough to be split leaves a left part that may be shorter than the threshold, which is stated as it is, and the span policy merges it later. A third way, shadowing part of a span with an entry of its own, is left out of the format.

43.2.3 Zeros

A run of zeros among the new bytes, long enough that a `Zero` entry is cheaper than the bytes, becomes a `Zero` span: 32 bytes is a reasonable cut. Encodings without padding hold few zeros, but `Fixed` values and large preallocated arrays have them.

43.3 The span policy

Two policies bound the description, and they combine: one bounds it in the leaves at a constant cost per edit, the other drives it toward nothing where the sequence has gone cold; spans whose entries sit in the header are exempt from both.

- **Bounded spans**, the discipline of a B-tree's leaves without the interior nodes: no two neighbouring spans whose entries lie in leaves are shorter than `L` together. A flush keeps the rule after its fold has been mapped to spans and before it places anything: it writes the spans that break it again as one, among its own new bytes, which costs at most `L` bytes per boundary removed and never cascades, since a merged span is longer than either neighbour was. The mean span in the leaves is then at least `L / 2` long, everywhere and at all times.
- **Merging by temperature**: a run of spans whose content has not been written for a while is written again as spans of up to a page, by the rotating walk. Each boundary is removed once, after the edits that made it, and a cold span of a page costs about six bytes, 0.15 % of what it shows. It needs a temperature per span, which heat gives.
- **The header is exempt**: spans of any length may keep their entries there. An edit site that every flush touches — typing, dragging a shape, moving a slider until a setting sounds right — splits its spans while it is hot, and merging them would write the same bytes again at every flush, only for the next edit to split them anew. The header can afford them, since it holds the hottest entries, is written by every flush anyway, and holds at most a page of entries, so the exemption adds at most a page to the description, however large the file.

The bound takes effect where entries leave the header: when the header would overflow, the flush chooses its coldest runs of spans to evict to leaves, by heat, and merges each run's short spans as it evicts them, so their bytes are written again once, after they have gone cold. Since merging writes bytes, the flush makes that choice before it places anything, from entry sizes it estimates without encoding them.

The first policy gives the worst-case bound of the guiding principle, the second the low overhead of a large file that is mostly cold, and the exemption keeps the first from charging hot edits for merges that the next edit undoes. A larger `L` makes the description smaller and each edit more expensive. Neither touches the format, which allows spans of any length from one byte to a page.

The constants, as measurements chose them for kladde-rs: the bound `L`, 64 bytes; the length below which a `Ref` cut by an edit is rewritten rather than split, 64 bytes; the inline threshold, 64 bytes; the flushes after which a span counts as cold for the walk, 16; and the walk's price μ, 0.02 entry bytes per byte written. They were measured on a handful of drawings and texts, and remain open questions, as do what should count as cold, and how hot spans and the header should share the span table's room.

43.4 The `Inline` threshold

The format fixes the ceiling at 126 bytes; the policy threshold belongs below it.

Below roughly the encoded size of a `Ref`, inlining is strictly better — the address would be as large as the data. Between there and the ceiling it is a trade: inline payloads ride the header for free while hot but inflate the table pages when cold, whereas a `Ref` plus its data bytes is written once but occupies room

in a data page and keeps that page partially live. The accounting argues for keeping the threshold low: an inline payload is charged to the table page that holds it and is only reclaimed by rewriting that page, whereas a `Ref`'s bytes are charged to a data page that consolidation can clean apart from the entry. 64 bytes is a good default.

43.5 Placement

43.5.1 Cutting content across pages

A flush writes whole pages, and a `Ref` shows bytes of one page only, so new bytes are cut into spans that each fit in one page.

- **A run longer than a page becomes whole pages first**, each one span, plus a remainder; that is the one cut that is forced rather than chosen.
- **The remainders are packed in sequence order with a bounded look-ahead**, pulling consolidation's victims into any page with room its own content cannot use, and cutting only to keep a page from closing more than `0` empty; packing is described with consolidation, since what it has to know is consolidation's side of the interface.
- **Whoever writes bytes may cut them anew**: span boundaries carry no meaning beyond "these bytes lie contiguously here", so a flush re-cuts what it writes at will, and consolidation may split or merge spans it relocates.

Free re-cutting removes the worst case of fixed-size pieces: chunks all just over half a page could otherwise never share a page. The price of each cut is one entry, and in ordinary flushes the packer prefers slack to a split.

A flush writes every span it produces **whole**; the old bytes remain behind as garbage until consolidation reclaims them.

A large payload is written twice: once into the journal when its record is appended, and again into data pages when the flush places it — a pasted image, say, or a document the application built outside any handle and brought in whole. A flush could instead state spans over the payload's bytes where they already lie, in the journal's pages, turning those pages into data pages, which would halve what a large paste writes; it would need journal pages that hold whole pages of a payload aligned to the data pages' content, and the reuse rule to hold for pages that change kind, and is an open idea.

43.5.2 Page contiguity buys nothing

Adjacent pages need exactly as many entries as scattered ones, because a `Ref` shows bytes of one page only, so a bulk write of many pages is one span 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.

43.5.3 Where the pages come from

The reusable pool is an ordered set, a list, and a watermark, because the 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();
```

Taking a page takes the **lowest** page in `ready`, and extends the file only when `ready` is empty; journal pages and payload pages come from the same pool. Lowest-first lets pages at the end of the file stay dead once they die, so that truncation can return them.

43.6 The header as write buffer

The root of the span 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 span table's **write buffer**: every flush states the header's entries again, and fills the header with the hottest entries before any leaf, at zero additional page writes.

- The common small flush writes **no** table page beyond the header, and no data page when its new bytes are all inline.
- An entry carried in the header is restated every flush at the new epoch, for free.

When the header overflows, the coldest entries go to leaves, ranked by heat. What the header can hold is bounded by `MAX_PAGE_CONTENT`, so the ranking covers a few hundred entries however large the file is, and a sort of them on the rare flush that overflows is beneath measurement. Evicting the coldest runs of spans contiguous in id, rather than the coldest entries wherever they sit, keeps the leaves' ids dense, which keeps their delta encoding short, and lets the evicted run be merged to the bound as it leaves.

43.7 The shape of the tree

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 span table, nothing descends the tree by key, so there is no ordering invariant to maintain across pages.

- While everything fits in the header: **zero extra pages** — the steady state of every small file.
- Overflow goes to leaves referenced directly from the header, with entries continuing to share the header alongside the child references. At one to two bytes per child reference, a header can reference two to four thousand leaves, one to two million spans.
- Beyond that, one layer of interior pages reaches some 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. A depth change is a single-page event: when the header's child list outgrows its room, move the whole list into one fresh interior page and reference that instead.

43.8 The cut

The cut writes every stated span's entry: the header's into the header, the rest into new leaves, in id order, each leaf closed when the next entry does not fit, so that every leaf but the last is full to within one entry. The last leaf, or the header when there is no leaf, takes **fillers**: whole sparse leaves whose live entries fit its room, restated there, which drains them for free (the page rewrite). 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 the flush adds and unlinks, so the header's share is chosen against an estimate and corrected: if the list turns out longer, the header gives up its coldest entries to the leaves.

Entries are derived from the span table as it finally stands, after everything else in the flush, so each span is stated exactly once in the flush's epoch, as the format requires, however many sources touched it.

43.9 Correctness: the differential oracle

Build the oracle before the optimizer.

Test the store on its own against a plain byte vector: any series of `Write`, `Splice`, `Move` and `Paste`, flushes, reopens and simulated power cuts must leave the sequence holding the vector's bytes. Above the store, **after every operation, the sequence must equal the canonical encoding of the in-memory root value**, byte for byte. Randomized operation series with shrinking.

43.10 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 `E` is partially written, not fsynced.
- All `Data` and `Table` 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 `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 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 44

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, and merges short spans so the description stays small. 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.

44.1 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 a *region of the sequence*: how many spans it takes to show its bytes. Runs of short spans arise where edits happen, since every splice and every split leaves a boundary; they cost entries in table pages, memory in the span table and the rope, and time at open; and no page-level counter can see them, because a run of short spans whose entries sit densely in one full leaf makes that leaf look perfect.

Page fragmentation is a property of a *page*, `Data` and `Table` alike: how much of it current state still relies on. It is measured by coverage against capacity; it costs file size and nothing else; and nothing that counts spans can see it, because long spans can lie thinly spread over sparse pages.

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, 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 one direction: more spans need more table pages, and merging spans writes their bytes again, leaving the old ones as data-page garbage.

Only page fragmentation is urgent, and that asymmetry 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 bounded already by the span policy's merges as entries leave the header, and what remains is a standing debt that does not grow while nobody writes the region — so a rotating walk pays it down *eventually*, 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.

44.1.1 Three 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 spans, through the reverse index	the spans' bytes, into data pages, each span restated with its new address
Page rewrite	page fragmentation of `Table` pages; outer fragmentation, in compaction mode	the sparsest table pages, one victim at a time — the highest first, in compaction mode	the victim's live entries, from the span table	those entries, restated at the cut
Rotating walk	description fragmentation	a cursor that rotates through the sequence	the bytes of the runs it merges	one span per merged run, among the flush's new bytes

All three end in the same place — fresh pages plus fresh entries — which is what lets one placement pass serve all of them.

Every one of them is simpler than it would be over allocations, because a span's id never changes: evacuation writes a victim's bytes elsewhere and states each span again with its new address, and nothing else changes; a page rewrite states each live span in it again under its own id, with no size to state again and no tombstone to carry; and merging a run of spans materializes no zeros, since no span has gaps.

44.2 Evacuation

Evacuating a data victim writes the bytes of its reached spans into new pages and states each span again with its new address. The victim's spans come from the reverse index, exactly and in `O(1)` each. A span is all or nothing — its bytes move together, since a `Ref` shows one contiguous run — except that a span may be cut at a page boundary where packing needs it, its tail becoming a new span that leads the next page. Neighbouring short spans in a victim can be merged in the same pass, at no extra write, since their bytes are being written anyway.

44.2.1 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: 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 superseded 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. It **fails gracefully**: under a workload with no locality at all, the age term carries no information and the policy degrades to plain greedy, which is the right fallback anyway.

44.2.2 Scoring a data page

Take LFS's cost-benefit ratio unchanged, with `u = coverage[p] / capacity` the live fraction and `age = current_epoch - page_epoch[p]`:

```
score(p) = (1 - u) · 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 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 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.

44.2.3 Victims are pulled one at a time

A victim is taken only when all of its spans have somewhere to go in this flush, since relocating only some of a page's spans reclaims nothing. In free filling that means the page's room holds all of them; on a budgeted page, one span 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) · C` bytes and writes `u_v · C`: all `C` of the victim's bytes come back, and `u_v · 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. **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 ≤ 1 / (1 + λ)`.

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`: free filling takes any victim that fits, and the churn floor does not apply to it.

`coverage` counts bytes shown by several spans once per span, so where they are, `u_v` over-states both what an evacuation writes and what it reclaims.

44.3 The page rewrite

A table victim is drained by restating its live entries elsewhere: each live span in it is stated again, under its own id, at the cut, and the victim then holds nothing live. Victims come from the table pages' buckets — sparsest first, sampled and scored as data pages are, or the tail first in compaction mode — and a victim is taken in one of two ways: into the room of the cut's last page, if all of its live entries fit there, which costs nothing, or under the budget, whose offers gather victims while their entries fit one leaf.

Its cost is those entries' encoded bytes, inline payloads included, which coverage already counts; its benefit is the whole victim. So it is priced exactly as evacuation is, `(C − written) / written`, and the two compete in the budget loop on equal terms, but for the fill floor, which holds evacuation's page alone, since the page rewrite opens none. Restated entries lie together in id order in their new leaf, so a page rewrite also repairs the delta encoding of entries that went cold far apart in the id space and were evicted together.

A victim with children — an interior page — is rewritten the same way, and its child references move with it; see unlinking.

44.3.1 Unlinking an emptied page

Nothing waits for a load. With entries, inline payloads, and child references all charged per byte, a table page whose last live entry is restated elsewhere reaches zero coverage in the same flush, and the quarantine does the rest.

One step stands between zero coverage and reuse: **an emptied table page is still named by its parent**, so it stays reachable from the header, and I2 forbids overwriting it 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. Deeper, the parent is rewritten without the child's reference, its other children carried over, and so is *its* parent, up to the header — the `depth − 1` path copy of a structural update. A writer must not rely on its own shallow trees here, since another writer may shape the tree differently, so:

- **The page table records each table page's parent**, which the tree property makes a single page number.

- **A child reference is charged to its parent's coverage**, and released when the child is unlinked; without that charge, a parent whose entries had all moved elsewhere would reach zero coverage while still holding the only reference to live children, and reusing it would orphan them.
- **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.

Data pages need none of this, having no parent.

44.4 Merging spans rides the rotating walk

A rotating walk over the rope looks at a fixed number of spans per flush, and merges runs of cold spans whose entries lie in leaves where the entry bytes the merge retires pay for the bytes it writes. The decision is per run and local: the walk visits the spans anyway, so the only question is whether a run is worth writing again as one span.

> **A merge pays when the entry bytes it retires exceed μ for every byte it writes.**

The data terms cancel, since the bytes the merged spans showed lose their coverage exactly where they lay, so a run's gain is the entries it retires, less the one it adds, less μ per byte. **The best runs are a maximum-subarray problem**: weigh each cold span by its entry's bytes less μ times its length, and Kadane's algorithm, in one pass over the spans the walk is traversing anyway, finds the maximal runs of positive weight, each cut into tiles of at most a page, since a `Ref` shows one page at most. A long span scores `entry − μ · len`, negative for any span longer than `entry / μ` — a few hundred bytes at `μ = 0.02` — so an already-long run is left alone, which is correct, since writing it again merges nothing and only moves bytes. A short, dense knot of tiny spans is exactly where the run of large positive weights is, so it is what Kadane returns.

Only cold spans take part, those whose bytes have gone unwritten for some number of flushes, since a run the application is still editing would be split again by its next edit, which would waste the merge; the heat of each span says how long. And only spans whose entries lie in leaves: the header is exempt.

The walk runs before placement, so a merged run's bytes join the flush's own new bytes, and are packed among them. μ is a price of the same kind as the churn floor λ — file bytes freed per byte written — and sits far lower, near 0, since what a merge chiefly buys is invisible in the file: an entry in memory per span retired, and the work of resolving it at every open.

The walk can also give a run of spans consecutive ids, so that their successors become implicit and their id deltas short, which shrinks entries without writing any data byte.

A new session resumes the walk where the last one left it, if the consolidator state says where, or else where open seeds it.

44.5 How a flush and consolidation compose

Every source changes spans as it goes, and entries come last; that gap is the whole interface between a flush and its consolidation. The fold, the span policy's merges, evacuation, the page rewrite and the walk all mark the spans they change as stated, and the cut derives every entry from the span table as it finally stands, so what the flush states is by construction exactly what it changed, each span once.

A relocated span is the same object as a fresh write: a run of bytes that has no place yet, differing only in whether its bytes are read from a data page or from the flush's new bytes. Placement cannot tell them apart, and does not need to.

44.5.1 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, while a survivor left behind costs a delay and nothing else. The walk's merges belong to the first stream, since they are chosen before packing, and once chosen they are writes like any other.

The elastic stream is 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 without ever opening a page it cannot fill.

44.5.2 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. Every cut is one more span, an entry and a node in the rope, which stays until a merge removes it, whereas slack is recovered by the next consolidation that picks the page up.

	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
splitting to fill it	one more entry and one more span in memory	a merge writes the run again as one span

Slack has no small bound of its own, which is what `θ` is for. Chunks all just over half a page are the adversarial case — no two can ever share a page, so if they were never cut, 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 − θ` full**, at a cost of at most one cut per page. On a page that holds content of the flush's own, that content is what gets cut, not a victim's survivor: the flush's own content is what the application is working on, so its cut is the likelier to be overwritten soon and take the extra span with it. A page opened only for consolidation has no such content, and there a survivor is cut instead, its tail leading the next page.

44.5.3 Packing in sequence order, with look-ahead

Pack in sequence order and fit by looking ahead, rather than first-fit over a size-sorted list. First-fit-decreasing packs tightly and destroys locality, which costs two things worth keeping: neighbouring spans in one page, which a later evacuation moves together and can merge, and consecutive ids with implicit successors.

```
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 sequence order, within reach.
        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 θ empty: cut the next chunk so the page closes full.
        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. **Step 2 fills the room with anything that fits and costs nothing**, on every page the last included: whole victims first — the tail first, in compaction mode — since each frees a page; then part of a victim too big to take whole, a whole span at a time, which frees no page yet but leaves the victim sparser.

44.6 The budget

The design has one throttle — a per-flush **maintenance budget** in pages written 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 λ, 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: **evacuation** relocates a data victim's spans and reclaims dead data bytes; **the page rewrite** restates a table victim's live entries and reclaims dead entry bytes. The rotating walk stays out of the ranking: what it retires is description, measured in entries and memory rather than in file bytes, and its price μ is its own floor.

44.6.1 The structures behind the ranking

Every source that starts from a *page* is ranked from **bucket queues**, which make "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 `coverage[p] * B / capacity`, so moving a page is a swap-remove from its old list, a push onto the new one, and a fix of the `at` entry of whatever the swap moved — which matters because coverage changes on every span stated or dropped, not once per flush. **Each page kind gets its own instance**, since data pages outnumber table pages by roughly the ratio of their contents; header pages belong in neither. A page in the sparsest non-empty bucket is within `1/B` of the true minimum live fraction, which is all a heuristic ranking needs. A page whose coverage reaches zero leaves its bucket at once, since it is retired at the next commit either way.

44.6.2 The budget loop

A flush and its consolidation run as one straight line:

```
fn flush(journal: Journal, budget: Budget) {
    // 1. The fold, mapped onto spans; the header's entries chosen, and the short
    //    spans of the runs that leave it merged; the rotating walk's merges.
    let fold = fold(journal);
    map_pieces_to_spans(fold);
    choose_header_and_merge_evicted();
    rotating_walk();

    // 2. Data pages: the flush's own chunks in sequence order, with victims riding
    //    along wherever a page has room its own content cannot use. See `pack`.
    let mut data = pack(new_chunks(), &mut elastic);

    // 3. The budget, in pages written, offer by offer to whichever page kind reclaims
    //    more per byte. An offer of data victims opens a page, and only one it fills;
    //    an offer of table victims opens none, since what it restates joins the
    //    leaves step 4 lays out, and costs the budget the pages those take.
    let mut spent = 0.0;
    while spent < budget.pages {
        let left = budget.pages - spent;
        let Some(offer) = [evacuation.offer_page(left), page_rewrite.offer(left)]
```

```
                            .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.opens_page() && offer.fill() < 1.0 - THETA { break; }  // the fill floor
        }
        spent += if offer.opens_page() { 1.0 } else { offer.written / C };
        offer.take(&mut data);
    }

    // 4. Table pages: the hottest entries in the header, the rest in leaves in id
    //    order, the last leaf's room to whole sparse leaves that fit.
    let table = cut();

    // 5. Unlink emptied pages, write, fsync, commit; then truncate a reusable tail.
    commit(data, table);
}
```

The two page kinds are combined, not traded off — but they do compete for the one budget, counted in the pages they write; 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 for an offer of data victims, and only when the offer fills it to `1 - θ`, which keeps a victim too many from costing a nearly empty page; only an offer holding compaction mode's tail is exempt. An offer of data victims 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 span cut at the page boundary, exactly as `pack` cuts the flush's own chunk.

An offer of table victims opens no page, so no fill floor holds it, and it costs the budget the pages its restatements take. The restatements of a sparse leaf take a fraction of a leaf, among everything else the flush states, so there is no page a floor could keep from closing short; the churn floor already weighs their bytes against the page they free. Held to the fill floor as well, leaves would stay half empty, since an offer would have to restate `1 - θ` of a leaf, which leaves less than half full could do only three or more at a time.

44.6.2.1 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× at `u = 1/2`, 5× at `0.8`, 10× 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.

So the two knobs 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 `live_bytes / (pages · C)` after each commit, over every page but the headers, and move the budget up while it sits below `τ`, down while above — multiplicatively, `budget · exp(4 · (τ - fill))` within its bounds, is a rule that works. Counting every page rather than only the live ones makes holes raise the budget as sparse pages do.

Ratio-greedy ranking starves costs that current ratios cannot see, and one such cost remains: description debt in a cold region, which never looks urgent, because nothing about it is changing. It is taken out of the ranking rather than weighted inside it: the rotating walk reaches every span in turn.

44.6.3 The complexity this incurs

- `O(r · log k)` in-memory work for a flush of `r` records over `k` spans, plus `O(log k)` per span stated, merged, or relocated under the budget;
- `O(1)` victim selection per candidate;
- page writes bounded by the flush's new data pages plus the budget plus the tree's depth;
- with the common small flush writing **no** table page beyond the header.

44.7 Two forms of consolidation

- **Free filling.** Victims, 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 cut's last leaf — so they cost no page write and compete with nothing. It runs to exhaustion.
- **Budgeted cleaning.** Beyond that, data pages that exist only to reclaim, offered one at a time and opened only when the offer fills them, and table victims whose restatements take leaves of their own. These compete, under the budget and the churn floor.

44.8 Outer fragmentation

Placement decides whether holes form, truncation returns what reaches the end, and a compaction mode moves what never would.

44.8.1 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 pool 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. Lowest first gives up the contiguity that a run from the end of the file would offer a large write, but contiguity buys nothing. Journal pages come from the same pool, so the journal, too, starts in the lowest hole rather than at the end, and so do payload pages, which become data pages where they lie.

44.8.2 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.

Truncating is cheap, but shrinking and regrowing the file on every flush of a workload that oscillates is not, so both reference implementations truncate only once the reusable tail exceeds a threshold, and at close and open.

An explicit full compaction would serve a file about to be exported or archived: flush, and keep flushing in compaction mode, with an unlimited budget, until no hole is left and the file ends at its last live page. Neither the reference algorithms nor kladde-rs offer one yet; closing after a few flushes comes close.

44.8.3 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.

An offer holding the tail passes the churn floor, and the fill floor if it opens a data page, 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. 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 removal, the pages it released sit in quarantine for two commits, so content moved then would land in pages the file grows by, which become the next tail, and the same content would move 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.

Free filling takes part in the mode too: while it lasts, `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.

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, is `O(1)` amortised.

44.9 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 — little garbage, and the file 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.

44.10 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: converging on a live fraction `τ` bounds the file at `live_size / τ`, at the price of the corresponding cleaning work. The bound is controllable and, in relative terms, independent of file size; the guiding principle adds the description's share.

44.11 Constants

kladde-rs's defaults, of which only the span policy's were measured:

- the target fill `τ`, 0.8, and the budget's start and bounds, 8 pages within 1 and 256;
- the churn floor `λ`, 1;
- the fill floor `θ`, 0.05, and the look-ahead, 16 chunks;
- the walk's length, 256 spans per flush, its price `μ`, 0.02, and the flushes after which a span counts as cold, 16;
- the victims sampled per pick, 8, and 16 buckets for each page kind;
- the hole share `h`, 0.25, and a reusable tail of 16 pages to trigger truncation.

Chapter 45

Consolidator state

What consolidation could carry from one session to the next in the private chain, and how it would tell whether what it finds there is still current.

Status: designed, not built. kladde-rs writes nothing to the private chain yet: every session starts its budget from its default, its rotating walk at the sequence's start, and its heat from the pages' epochs.

45.1 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:

- the budget would restart from its default at every open, and never converge on its target fill;
- the rotating walk would restart wherever open seeds it, rather than where the last session left off, and over many short sessions might never reach parts of the sequence;
- the heat of every span would be rebuilt from the epoch of the page that holds its entry, which the header re-stamps every flush, so every span stated in the header would look freshly written at every open, and the walk and the header's eviction would misjudge it.

45.2 What it would hold

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 walk	the span at which the next rotating walk begins	4 bytes
heat	for spans whose heat the file does not give exactly, the flush that last wrote their bytes	a snapshot, plus records appended since

Everything a flush puts in the state is known before its data pages are packed, since it is written together with them.

Heat is kept only where the file gets it wrong. A span's entry lies in a page whose epoch can only understate how long its bytes have gone unwritten, since content is never younger than the page that

holds its entry; that fallback is exact for a span written once and never restated. So a snapshot keeps mainly the spans whose entries consolidation or the header restated since their bytes were written. Heat records name an epoch and the span ids last written in that flush, sorted and stored as varint deltas, a byte or two per span, so a flush appends one record, one contiguous write; once the appended records outgrow the snapshot, a flush rewrites everything as a fresh snapshot, keeping only each span's newest mention.

45.3 Writing it

Every flush brings the state up to date, together with its own writes: it appends its heat record and rewrites the fixed fields. The state is ordinary content of the private chain, committed with the flush that wrote it, so a crash can never leave it ahead of or behind the file it describes.

A flush that finds no state it recognizes as its own replaces it, since the private chain is the only place to keep one.

45.4 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.**
- **Otherwise, a span's heat is current if no entry of the span is newer than `up_to_date`**, since its bytes cannot change without a new entry at the epoch of the flush that changed them; one that fails the test falls back to its entry's page epoch. Span ids are recycled, so a span stated after `up_to_date` may be another span under an old id, which the same test catches.
- **The walk's position and the budget are starting points either way**: at worst the walk resumes elsewhere, and the controller corrects a budget that no longer fits.
- **A state that cannot be parsed 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, so that records it could not vouch for do not become current by default.

Chapter 46

Transactions and batches

How application operations are grouped into the transactions the journal records, how they wait in memory, and when they are appended to the journal pages and folded.

Status: settled apart from the open questions at the end; kladde-rs implements it as described here.

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.

From the buffer on, this page is the design of the reference implementations, not one way among several to meet the format. kladde-rs and kladde-cpp each keep exactly one buffer of operations and the cursors into it described below, and append and flush by exactly the rules in Flush triggers. A departure — a second buffer, copying between spans, another trigger — is a deviation: it is recorded with its reason, in the implementation notes for kladde-rs and in kladde-cpp for kladde-cpp, and proposed as a change to this page if it is better, rather than taken as a free choice.

46.1 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 the largest pieces that fit in what is left of the journal's page budget, respecting nested transactions, and wraps each in a transaction	**optimization**: fewer and fuller journal writes, while keeping the journal within its budget	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 splices the element in and rewrites the count — and interrupting between them leaves a sequence that does not decode. 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. Kladde'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 pages, 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, so that each piece fills what is left of the journal's budget as far as the transaction boundaries allow.

46.1.1 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: build the value in memory, outside any handle, then insert it in one operation, which encodes it into one buffer and splices it in.
- 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.

46.2 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 (*shouldn't usually occur*)
… 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.

46.3 One buffer and three cursors

Every operation recorded since the last flush lives in one in-memory buffer `ops`, in the order it was recorded, whether it is already in the journal pages or not. Between two flushes, the operations form a single stream: some prefix of it has been appended to the journal pages, the operations on top of that are a batch's ready run, and those on top of that belong to the transaction that is still open. So everything

that happens between two flushes either appends to the buffer or moves a boundary between these spans; **nothing is ever copied from one span to another**.

The flush folds the operations the journal pages hold from this buffer, not from the pages, which only recovery reads back. Recovery decodes the journal's complete transactions into the buffer, and `open` folds them before it returns (see opening a file).

In kladde-rs, the buffer is a `VecDeque` of records, beside a byte arena, also a `VecDeque`, that holds their payloads. Positions in both are counted from the start of the session rather than from the buffer's front, so that discarding the folded prefix after a flush moves the front and nothing else: no record moves, and every position held elsewhere — a record's payload, a pending fragment of the flush pointing into the arena — stays valid. A journal's cuts are positions in it too: a paste names its cut by distance among the cuts of its journal, which the records keep in order, so no cursor moves them.

Three cursors, `front <= committed_cursor <= ready_cursor <= end`, divide the buffer into three spans:

- `committed := ops[front..committed_cursor]` — appended to the journal pages since the last flush: what the next flush folds;
- `ready := ops[committed_cursor..ready_cursor]` — a batch's operations, held back to be appended as one transaction;
- `transaction := ops[ready_cursor..end]` — the operations of the open transaction.

Each cursor also carries the encoded size of the records before it, so that the encoded size of any span is a subtraction, and deciding whether a span fits costs constant time. The journal's length prefix needs a transaction's size before its first byte is written anyway.

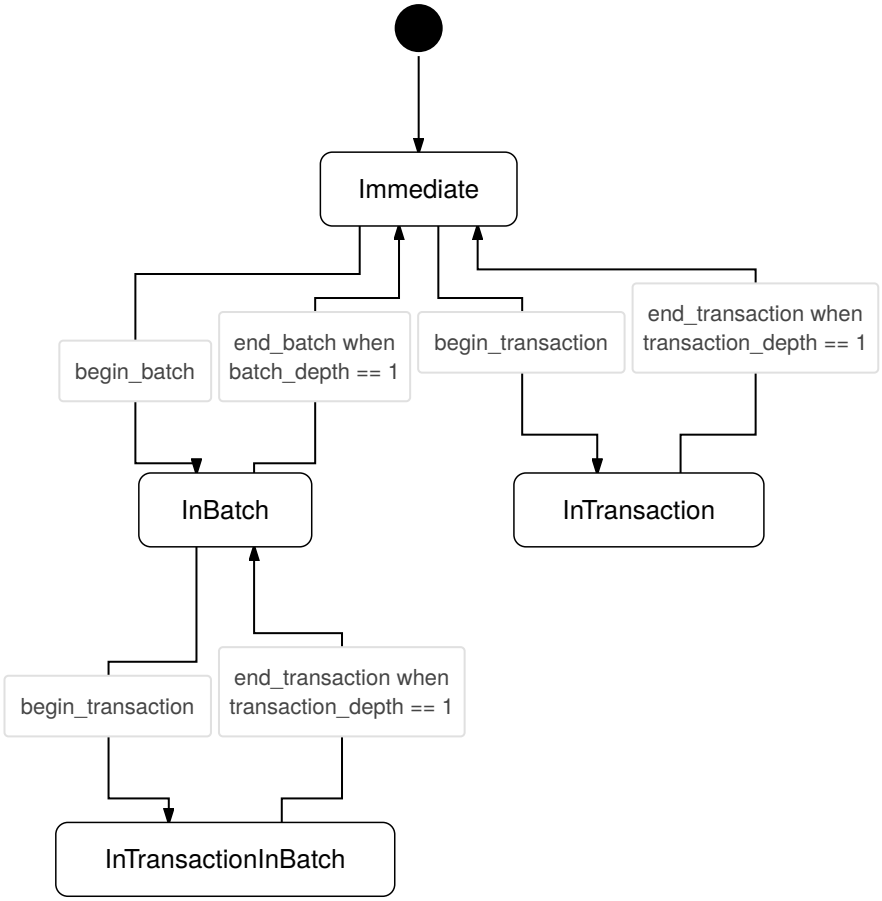
Moving the cursors is all that recording, appending, and flushing do:

- **Recording an operation** appends it at `end`.
- **Appending a span to the journal pages** encodes it as one transaction, writes it, and moves `committed_cursor` to the span's end.
- **A flush** folds `committed` and moves `front` to `committed_cursor`.
- **Discarding operations** — a transaction that failed halfway and is rolled back, or one the application abandons — moves `end` back to where they began.

The backend's state holds the depths of the open scopes and the cursors that are not implied by them:

```
enum TransactionState {
    Immediate,
    InBatch              { batch_depth: NonZero, committed_cursor: Cursor },
    InTransaction        { transaction_depth: NonZero, batch_depth: u32,
                           committed_cursor: Cursor },
    InTransactionInBatch { transaction_depth: NonZero, batch_depth: NonZero,
                           committed_cursor: Cursor, ready_cursor: Cursor },
}
```

`transaction_depth` and `batch_depth` count begins minus ends, with an implied zero in any variant where they are absent. `front` and `end` belong to the buffer, and the table below gives the cursors each state implies.



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.

46.3.1 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`	`end` (implicit)	explicit	explicit	explicit
`ready_cursor`	`end` (implicit)	`end` (implicit)	`committed_cursor` (implicit)	explicit
`begin_transaction()`	`InTransaction { t: 1, b: 0 }`	`InTransactionInBatch { t: 1, .. }`	`transaction_depth += 1`	`transaction_depth += 1`

	`Immediate`	`InBatch`	`InTransaction`	`InTransactionInBatch`
`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

The implied cursors are invariants at every operation boundary, not just labels: in `Immediate`, everything recorded is in the journal pages, and in `InBatch`, everything not yet appended is the batch's.

46.4 Flush triggers

> **Keep the journal within its page budget. A transaction that would not fit in what is left of the budget is preceded by a flush, and only a transaction too large for the budget on its own exceeds it, followed by a flush at once.**

The budget is a number of journal pages, `B`, and a transaction fits if the journal's stream, with the transaction appended, ends within `B` pages, counting each of the journal's payload pages as one. Each journal page holds `page_size - 8` bytes of the stream, the rest being its trailer, and a transaction of `n` record bytes takes `n` plus its length prefix plus its 4-byte `crc`. The writer takes the next journal page only once the stream continues past the current one, so fitting within `B · (page_size - 8)` stream bytes is fitting within `B` pages. A transaction's size is estimated with its bytes inline, which for large bytes is about the payload pages they will take. A transaction that does not fit even into an empty journal is **overlong**.

Whenever operations close — a bare operation recorded in `Immediate` or `InBatch`, or the outermost transaction ending, `InTransaction → Immediate` or `InTransactionInBatch → InBatch` — let `X` be the operations that just closed, `ops[ready_cursor..end]`, and apply:

1. **The candidate is `ops[committed_cursor..end]`:** `X` alone outside a batch, where `ready` is empty, and `ready` followed by `X` inside one, to be appended as one transaction.
2. **If the candidate does not fit, the batch is cut before `X`:** append `ready` as one transaction if it is not empty, which it always fits, and flush, unless the journal is empty. The candidate is now `X` alone, in an empty journal. A paste in `X` that names a cut of the journal just folded becomes a `Splice` of its value's bytes, since that cut has ended (moving values), and `X`'s size is computed again.
3. **Inside a batch, a candidate that fits stays in memory as the new `ready`**, which the batch holds back until its next cut or its end.
4. **Otherwise, the candidate is appended as one transaction, and if that took the journal past its budget, flush again.** Only an overlong `X` can do that, since step 2 left it an empty journal.

The other triggers:

- **The outermost batch ends, `InBatch → Immediate`:** append `ready` if it is not empty. It fits, so no flush follows.
- **An explicit `flush()`:** append `ready` if it is not empty, then flush — on an empty journal too, since a flush also consolidates. An open transaction stays in the buffer, past `committed_cursor`.
- **A flush when the application is idle:** `flush_if_journal_exceeds(fraction)` appends `ready` and flushes if the journal pages then hold at least that fraction of the budget, and does nothing otherwise; the check is a subtraction of two cursors. An interactive application calls it from its event loop's idle callback, so that the automatic flush, whose `fsync` blocks the mutating call that triggers it, rarely fires in the middle of a gesture. The fraction should stay well above zero: an idle flush ends the journal, and with it the cuts its records made.
- **Opening a file:** kladde-rs folds a non-empty recovered journal before `open` returns, since reads see the flushed state only. That also keeps the journal's rule that nothing is appended to a non-empty recovered journal, and no state outlives `open`.

So **an overlong transaction is flushed twice, once before it and once after it**, the first unless the journal was empty; and **a batch is always cut at the last boundary at which its run still fits**, which lies between two closed transactions or operations, never inside a transaction.

46.4.1 The journal's budget

`B` is 2% of the sequence's length in whole pages, at least 2 and at most 10, unless the application fixes it, and the writer computes it from the length as it stands whenever it checks a transaction against it. It is a whole number of pages, since a journal page once taken costs a page of the file however little of it is used.

A small budget keeps the file close to its content. At a flush, the journal it folds and the previous one, still the fallback, both occupy pages, and the pages a flush frees become reusable only after the next flush; a budget of `B` pages thus keeps up to about `2B` pages of journal beside the content. A budget of 256 pages, a megabyte, made the files of drawings of 40 KB to 1 MB up to 16 times their content, and a budget sized to them kept them within 1.2 to 2.2 times.

Every flush is a chance to consolidate, so flushing often costs little beyond the flush's `fsync`: a flush folds a journal of a few pages in one to two milliseconds, within a frame of an interactive application, where a journal of a megabyte stalled the edit that triggered it for 46 to 77 ms.

The floor of 2 pages keeps a small file from flushing at every few edits: a budget of 1 page flushed the same drawings two to four times as often as 2 to 4 pages and made their files no smaller. **The ceiling of 10 pages** keeps a large file's journal, and with it the pages it holds and the time a flush takes, bounded; a file of tens of megabytes gains from a longer journal only fewer flushes, which an application that wants them can have by fixing the budget.

46.4.2 Payload pages

When a transaction is appended, the bytes of each of its `Write`s and `Splice`s that carries at least `payload_threshold` bytes, 2 KB, are written into payload pages first, and the transaction names them rather than holding them. The decision falls at the append, not when the operation is recorded, because the records of an open transaction or a batch's ready run may still be discarded, and the cuts that an application holds name records by their number in the buffer, which splitting a record there would change. So the buffer keeps each record as it was recorded, its bytes in the arena, and records beside the arena which runs of it also lie in a payload page, at which page, offset and checksum; the encoder writes such a record as one record per run, and the flush maps the runs onto the pages. Nothing above the store sees a difference: a container hands the store its bytes as before.

- **The bytes go into the journal's open payload page as far as it has room, and the rest into new pages, a page's worth at a time**; a last part shorter than the threshold that would open a page stays inline in the journal, in a record of its own. The pages a transaction touched are written whole, framed with the journal's epoch, before the transaction is appended, and the open page is written again whenever a later transaction of the journal appends to it.
- **A record whose bytes are cut into runs becomes one record per run**, the first removing what the record removed and the rest removing nothing at the cursor, delta 0; a `Write` becomes such splices too, the first removing what it overwrote, so that it still makes one cut, and a paste of the overwritten value still names one.
- **Payload pages come from the reusable pool, lowest first, as journal pages do, but leave its last page alone**, growing the file instead: a large transaction is followed by a flush, which seals only if a reusable page is left for the next journal to begin in, and without its seal that flush could not commit on a worker, and a lost header would lose what was appended after it.
- **Payload pages count toward the budget**, so a transaction with more payload pages than the budget is overlong: preceded by a flush if the journal is not empty, and followed by one at once, which adopts its pages and writes a header and a leaf or two. The budget so bounds how much content waits for a flush to be durable under a power cut, as well as the journal's share of the file.
- **A transaction that is abandoned after its pages were written** leaves them in no world: they return to the pool, and the bytes it added to the open page are dead from the start. A payload page that cannot be written fails the transaction and poisons the store, as a journal append that cannot get a page does.

The threshold is about half a page, and unmeasured: a payload page holding `r` bytes costs a page write now and leaves `4081 - r` bytes of slack until an append or consolidation fills it, where inline bytes cost `r` in the journal and `r` again in a data page the flush shares with other content. Creating a file of 4.26 MB, one splice of the whole drawing, then takes four journal pages besides its payload pages, and the file comes out at 1076 pages, within three pages of a consolidated copy, where with every byte in the journal it took 2139.

46.4.3 Why `ready` always fits

`ready` only becomes longer in step 3, which checked that it fits after what the journal pages already hold. Nothing else is appended while a batch holds a ready run, since every closing operation inside the batch either joins `ready` or cuts the batch, and the cut appends `ready` first. So what the journal pages hold does not change until `ready` itself is appended, and `ready` still fits then.

46.4.4 Why flush first

A flush before the append leaves no journal page behind that only the start of an overflowing transaction occupies. Appending first and flushing once the journal is over budget would end every journal with such a page, nearly empty; and at the moment of a flush, the journal it folds and the previous one, still the fallback, are both off limits, so that it would cost two pages at a time.

It costs nothing. The operations of `X` are already in the buffer, and their encoded size is known from the cursors, so deciding before the append is as cheap as deciding after it. A flush folds only `committed`, so `X` waits in the buffer past `committed_cursor` and is appended to the next journal; only the order of the two steps changes.

The flush after an overlong transaction keeps a long journal from lingering: the journal is back within its budget as soon as the operation that exceeded it returns.

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 or payload pages in proportion too once it is appended.

46.4.5 Consequence for write-phase state

A flush may run while operations are still in the buffer past `committed_cursor`: the `X` that step 2 holds back, or, at an explicit `flush()`, an open transaction. The write-phase state — the sequence's length and the journal's cuts — describes them as well, so a flush leaves it as it was, and the flush folds only the records before `committed_cursor`, whose positions refer to the sequence as it stood before the held-back ones.

Discarding operations moves the length and the cuts back to what they were where the discarded operations began.

46.5 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 whose transaction would not fit in the journal's 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: handles hold positions in the

sequence, which a flush does not change, since it rewrites only where the sequence's bytes lie on file. 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.

46.6 A flush off the caller's thread

An automatic flush that sealed its journal may hand its `fsync` and its header write to a thread of their own, and the mutating call that triggered it then returns without waiting for the disk. Folding a journal and writing a flush's pages take well under a millisecond for files of the sizes kladde is meant for, while the `fsync` takes milliseconds on an SSD and more on a slow disk, which an interactive application feels as a stall in the middle of a gesture. So the calling thread seals, folds, places and writes the pages as before, and a worker does what remains: the `fsync`, then the header write. The application goes on appending to the next journal meanwhile, which the seal makes reachable from either header, so an application crash before the header is written loses nothing that returned. Doing it is optional; the format requires only the seal, and a flush that runs entirely on the calling thread writes the same file.

The worker shares nothing with the calling thread but the file, through a second handle to it, and one hand-over:

- **The next flush waits for the worker before it seals**, and so do an explicit flush, before it returns, and closing the store. One commit is in flight at a time.
- **The pages the commit before freed become reusable only when the worker reports**, since its `fsync` is what retires the world that held them (the reuse rule); until then neither the next journal nor anything else may take them.
- **A flush that cannot seal syncs on the calling thread**: without a seal, transactions appended during the `fsync` would belong to a journal only the unwritten header names. A flush seals only where a reusable page can start the next journal, since a page added at the end of the file before the flush's own pages would keep the file from being truncated to them.
- **A failure on the worker poisons the store** when it is reported, at the next flush that waits for it, as a failure on the calling thread would at once. Reads do not wait: the pages they read are in the file already, synced or not. A failure reported only as the store is dropped has nobody left to tell, and is lost, which is no worse than a power cut at that moment.
- **A session's first sealing flush syncs the page it seals to before appending the seal**, as the specification requires; that one extra `fsync` per session runs on the calling thread.

An explicit `flush()` keeps its meaning — everything before it is durable when it returns — by waiting; only the flushes that mutations trigger run their commit off the calling thread. kladde-rs offers this behind an option, off by default, until the Inkscape demo has shown whether the idle flush of flush triggers already keeps the stall out of gestures.

46.7 Open questions

- The journal's page budget: whether it should adapt to an application whose transactions routinely exceed it.

Chapter 47

kladde-rs

The Rust implementation of kladde: one of the two reference implementations of the specification, with kladde-cpp in C++.

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.

47.1 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 durable 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.

47.2 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::Durable;
use kladde_types::DurableVec;

#[derive(Durable)]
struct Notes {
    title: String,
    lines: DurableVec<String>,
}

let mut notes = Kladde::new(Notes {
    title: String::from("today"),
    lines: DurableVec::new(),
});

notes.get_mut().lines_mut().push(String::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.

47.3 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 `Mut`.

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 `Muts` at all.

47.4 The documents

document	contents
Crate layout	the workspace, the dependency direction, and where the seam falls
Durable and Mut	the central traits, regions, `Fixed`, why mutation needs a `Mut`, and why it returns a `Result`
Containers	how the built-in containers are encoded and indexed, and cuts
The derive macro	what it generates, fixed forms, enums in place of trait objects, and why `Muts` are generated per type
Transactions	the RAII 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

47.5 Status

A prototype of the whole design exists; nothing is frozen. Where the implementation departs from the reference algorithms, or found them unclear, its implementation notes say so.

area	status
Schema descriptors, encoding, fingerprints, rules of nesting	implemented and specified
The encoding of values; schema binding to Rust types; fingerprint check at open	implemented
Containers, derive macro, `Muts`, regions, `Fixed`	implemented
File format, span table, journal, recovery	implemented, with a span table of two levels
Flush, span policy and consolidation	implemented
Transactions, batches, and cuts	implemented; cuts on the vector only
The private chain	written and read, but nothing kept in it
Schema evolution	designed, not built

Chapter 48

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, map, string, blob.
3. **Deriving your own types** — structs, enums, and enums in place of trait objects.
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. **Implementing `Durable` by hand** (*advanced*) — opaque types, and implementing the trait by hand.

48.1 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(total size)` per mutation. Kladde is `O(size of the change)`, at the cost of a more constrained type vocabulary.

48.2 Status warning

Kladde is early: the file format is a draft, and a file written today may not open with a later version.

Chapter 49

Getting started

49.1 The mental model

A durable data structure lives in two places at once.

In memory, it is an ordinary Rust value: a `DurableVec<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.

49.2 Your first durable value

```
use kladde::{Kladde, Durable};
use kladde_types::DurableVec;

#[derive(Durable)]
struct Journal {
    owner: String,
    entries: DurableVec<String>,
}

fn main() -> kladde::Result<()> {
    let path = std::env::temp_dir().join("getting-started.kladde");
    let mut journal = Kladde::create(
        &path,
        Journal {
            owner: String::from("ada"),
            entries: DurableVec::new(),
        },
    )?;

    journal
        .get_mut()
        .entries_mut()
        .push(String::from("first entry"))?;
```

```
    println!("{} entries", 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 `Mut`, no file access, no cost beyond the read itself.

Writing goes through `get_mut()`, which hands you a `JournalMut`. Every field of a derived type gets a `_mut()` accessor on its `Mut`, which reborrows the same backend one level deeper. So `journal.get_mut().entries_mut()` is a `DurableVecMut`, 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.

49.3 Why the `Mut` 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 `Mut` is the substitute. It stands where a `&mut` would — `get_mut()` hands it out as `Vec::get_mut` hands out a `&mut T`. It is an RAII token that proves you have exclusive, recording access, and it carries two things a plain `&mut` could not: the backend to record into, and the *region* of the file's content that this particular field occupies — where its encoding starts in the file's one sequence of bytes, and how long it is.

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.

49.4 Nesting

`Mut`s compose to arbitrary depth, and each level reborrows the same backend:

```
let mut journal_mut = journal.get_mut();
let mut entries = journal_mut.entries_mut();
entries.push(String::from("second"))?;
entries.push(String::from("third"))?;
```

Or in one chain, when you only need one mutation:

```
journal.get_mut().owner_mut().set("grace")?;
```

Each `_mut()` borrows the `Mut` it is called on, so a `Mut` hands out one field's `Mut` at a time. To mutate several fields together, `parts()` hands out a `Mut` for every field at once, in a struct named after yours:

```
let mut journal_mut = journal.get_mut();
let JournalParts { mut owner, mut entries } = journal_mut.parts();
owner.set("grace")?;
entries.push(String::from("handed over"))?;
```

Hold a `Mut` only as long as you need it. It borrows the root, so nothing else can touch the structure while it is alive.

Chapter 50

The built-in containers

Kladde has durable text, two vectors, a map and a blob. Text and vectors of fixed-size elements are the standard `String` and `Vec<T>`, durable as they are, since they need nothing beyond what the standard types hold. `kladde-types` provides the other three, `DurableVec<T>`, `DurableMap<K, V>` and `DurableBlob<T>`, since each keeps an index or a serialization of its own in memory. Every container's content lies inline in the file's one sequence of bytes, wherever its value lies, so a short list costs no more on file than its elements and its count.

50.1 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 `Durable`/`Mut` 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.

50.2 `String`

Growable text.

```
let mut owner = journal_mut.owner_mut();
owner.set("ada")?;
owner.push_str(" lovelace")?;
```

On file, the text's UTF-8 bytes and then the byte `0xFF`, which UTF-8 never contains; in memory, the `String` itself. Text states no count, so every edit — `push_str`, `insert_str`, `replace_range`, `truncate` — is one splice of the file's sequence, wherever it lands and however long the text is.

`set` records only what changed: it compares the new text with the old one and splices in the bytes between their common prefix and suffix, or nothing if they are equal. So you may set a whole text at every change — a path's data rewritten at every step of a drag, say — and the file records a few bytes rather than the whole text.

Text is UTF-8; for a byte buffer, use a `Vec<u8>`.

50.3 `Vec<T>` and `DurableVec<T>`

A growable sequence, in two types with the same encoding: **`Vec<T>` for elements of a fixed size, and `DurableVec<T>` for any.**

A `Vec<T>` is durable when every value of `T` encodes to the same number of bytes — `u8`, `f32`, a tuple or struct of them, or `Fixed<T>` — since then it finds element `i` by multiplication. For any other `T`, such as `String`, using a `Vec<T>` as a durable value fails to compile, and the message names `DurableVec<T>`:

```rust
#[derive(Durable)]
struct Journal {
    entries: Vec<String>,  // a `Vec<T>` is durable only for a fixed-size `T`
}
```

`DurableVec<T>` takes any `Durable` `T`, and nothing more — no `Clone`, no `Serialize` — and adds cutting and reordering:

```rust
let mut journal_mut = journal.get_mut();
let mut v = journal_mut.entries_mut();
v.push(String::from("a"))?;
v.insert(0, String::from("b"))?;
v.get_mut(1).unwrap().push_str("c")?;  // the `Mut` of one element
let first = v.remove(0)?;   // yours now, as a plain value
v.push(first)?;             // encoded again, at the end
let moved = v.cut(0)?;      // removed, its bytes kept for a paste
v.push_cut(moved)?;         // pasted at the end, without writing its bytes again
v.move_element(0, 1)?;      // reordered, without writing an element's bytes
v.delete(0)?;               // removed and dropped
```

A `Vec<T>`'s `Mut` offers the methods they share: `set`, `insert`, `push`, `remove`, `pop`, `clear` and `get_mut`. Reading needs no `Mut`: both dereference to `[T]`, so `len`, indexing and `iter` work on the in-memory elements.

Encoding. The number of elements as a varint, then the elements, back to back; a `Vec<T>` and a `DurableVec<T>` of the same elements have the same bytes and the same schema. In memory, a `DurableVec<T>` of elements whose size depends on their value keeps a Fenwick tree of their sizes beside its `Vec<T>`, four bytes per element, so that the `Mut` of element `i` finds where it starts in `O(log m)`; for fixed-size elements it keeps no tree.

Cost. A push, an insertion and a removal are each a splice of the element plus a rewrite of the count, in one transaction; a change of an element's size is a splice inside it; and `move_element` writes no element's bytes at all. None of them rewrites the elements behind it: the file records where its bytes lie as linked runs, so bytes that shift are not restated.

Moving. `remove` and `pop` hand the element back as a plain value; pushing it elsewhere encodes it again. `cut` removes it too, but keeps its bytes as a *cut*, which `push_cut` or `insert_cut` pastes into any `DurableVec` of the same element type without writing them again: the move costs a few entries in the file however large the element is. A cut lives as long as the journal that made it: you may hold it across transactions, and a paste after the next flush still works, but encodes the element again, as `push` would. `Cut::is_alive()` tells you whether a paste would still write none of its bytes; dropping a cut unpasted drops the element. `delete` and `clear` remove and drop.

50.4 `DurableMap<K, V>`

A key-value map that keeps its entries in insertion order. `K: Eq + Hash + Durable`, `V: Durable`.

```rust
let mut contacts = book_mut.contacts_mut();
let ada = String::from("ada");
contacts.insert(String::from("ada"), contact)?;
contacts.get_mut(&ada).unwrap().email_mut().set("ada@example.com")?;
contacts.delete(&ada)?;     // removes and drops; `remove` hands the value back
```

Reading — `get`, `contains_key`, `iter`, `len` — needs no `Mut`. Inserting under a key that is already present replaces its value and returns the old one, as `HashMap::insert` does.

Encoding. The number of entries, then each entry's key and value, in the map's order; the schema describes it as a map of its key and value types, so a tool can show it as one. The file does *not* support key lookup at all — that is what the in-memory map is for — so it needs no hashing structure on file. In memory, the entries sit in a vector, in their order on file, beside a hash table of their indices, so each key is stored exactly once and `K: Clone` is not required.

Removal swaps: the last entry moves into the removed one's place, without writing its bytes, so the file stays dense, and an insertion appends.

Keys are immutable. No `Mut` is ever handed out for a key; `Durable` on `K` governs only how its bytes are encoded.

50.5 `DurableBlob<T>`

An escape hatch for foreign types.

```
#[derive(Durable)]
struct Config {
    window: DurableBlob<WindowGeometry>,  // a foreign serde type
}
```

Wraps any `T: Serialize + DeserializeOwned + Default` as an opaque, serialized payload, prefixed with its length. Use it when a type is not yours to change and cannot get its own `Durable` implementation.

It is deliberately the worst option. A blob is serialized again in full on every change, which costs time in proportion to its size, though only the bytes of the serialization that changed are recorded; 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.

50.6 What is missing

A **rope**, for large text with efficient middle-insertion, is designed but not built.

Tuples of up to twelve components are durable, encoded like a tuple struct, and their `Mut`'s `parts()` hands out a `Mut` per component. `Option<T>` is durable as an enum, and its `Mut` offers `set`, `take` and `as_mut`; `Box<T>` is persisted as the `T` it holds, which is how a recursive type gets a size in Rust.

50.7 Choosing

you have	use
a sequence of fixed-size values	`Vec<T>`
a sequence of anything else, or one you cut and paste from	`DurableVec<T>`
text	`String`
bytes	`Vec<u8>`
a lookup table	`DurableMap<K, V>`
a value that changes often and should keep its size	`Fixed<T>`
a few values that belong together	a tuple, or a derived struct
your own struct or enum	`#[derive(Durable)]`
a foreign type you cannot change	`DurableBlob<T>`, reluctantly
a foreign type whose layout you know	a hand-written impl

Chapter 51

Deriving your own types

`#[derive(Durable)]` turns your own struct or enum into a durable type, as long as every persisted field is itself `Durable`.

51.1 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 `Durable` trait and an error message that prints the expected and found types identically.

Here is a complete program:

```
use kladde::{Kladde, Durable};
use kladde_types::DurableVec;

#[derive(Durable)]
enum PhoneNumber {
    Mobile(String),
    Landline(String),
}

#[derive(Durable)]
struct Contact {
    email: String,
    phones: DurableVec<PhoneNumber>,
    starred: bool,
}

fn main() -> kladde::Result<()> {
    let mut contact = Kladde::new(Contact {
        email: String::from("ada@example.com"),
        phones: DurableVec::new(),
        starred: false,
    });
```

```
    let mut contact_mut = contact.get_mut();
    contact_mut.starred_mut().set(true)?;
    contact_mut
        .phones_mut()
        .push(PhoneNumber::Mobile(String::from("555-0100")))?;
    Ok(())
}
```

The `Durable` in that `use` is the derive *macro*. It shares its name with the `Durable` *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.

51.2 Structs

The macro generates a `Mut` type — `ContactMut` — with a `_mut()` accessor per field, and a `set` that replaces the whole value:

```
let mut contact = /* ... a ContactMut ... */;
contact.starred_mut().set(true)?;
contact.phones_mut().push(number)?;
```

It also generates a `ContactParts` struct, and a `parts()` method on the `Mut` that returns the `Mut`s 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 is its fields' encodings, back to back, inline in whatever contains it. A field's `Mut` finds where the field starts: at a constant offset up to the first field whose size depends on its value — a varint, text, a container — and after that at the sum of the encoded sizes of the fields before it, which the in-memory values know. Integers wider than a byte are varints, so a `u32` takes one byte below 128 and up to five above.

That includes primitive fields. `bool`, `i32`, `char` and friends all implement `Durable`, specifically so the macro can treat every field uniformly rather than special-casing leaves. So do tuples of up to twelve `Durable` types, encoded like a tuple struct, `Option<T>`, `Box<T>`, which is persisted as the `T` it holds, `String`, and `Vec<T>` of fixed-size elements.

51.3 Fixed-size values

`Fixed<T>` keeps a value at the same size whatever it is: every integer at its full width, every `char` in four bytes, and every enum with its discriminant at a fixed width and its variant padded with zeros to the largest. Use it for a field that changes often, such as a counter, which would otherwise change size whenever it crosses a power of 128, and for the elements of a large vector, which then finds element `i` by multiplication, keeps no index of where its elements start, and may be a plain `Vec`.

A derived type can be wrapped in `Fixed` only if it is marked `#[kladde(bounded)]`, which requires every field's type to have a fixed form too:

```
#[derive(Durable)]
#[kladde(bounded)]
struct Shape {
    x: u32,
    y: u32,
    kind: ShapeKind,        // also #[kladde(bounded)]
}

#[derive(Durable)]
struct Drawing {
    shapes: Vec<Fixed<Shape>>,   // eight bytes per shape, plus the kind's
```

```
    revision: Fixed<u64>,
}
```

`Fixed<T>`'s `Mut` replaces the value whole, with `set(Fixed(...))`: changing one field means building the whole value, and writing all of it, which for a fixed-size value is an overwrite in place. Types holding text, a container, or themselves have no fixed form, and wrapping one in `Fixed` fails to compile with an error that says so. A generic type marked `#[kladde(bounded)]` has a fixed form wherever its parameters do.

51.4 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(Durable)]
#[kladde(transparent)]
struct Meters(u32);

fn walk(distance: &mut Kladde<Meters>) -> kladde::Result<()> {
    distance.get_mut().get_mut().set(42)
}
```

Its `Mut`'s `get_mut()` returns the field's `Mut`.

51.5 Enums

An enum is its discriminant, as a varint, followed by the selected variant's fields, each variant encoded like a struct in its own right. So a value takes the size of its own variant, not of the largest.

In the schema, every variant holds exactly one value: a unit variant holds `()`, a variant with one unnamed field such as `Mobile(String)` holds that field's type, and any other variant holds a struct of its fields, named after the variant, its positional fields named by position, `"0"`, `"1"`. So moving a variant's fields into a struct of their own changes neither the file nor the fingerprint: `R { w: u32, h: u32 }` and `R(Rect)`, with `struct Rect { w: u32, h: u32 }`, are the same variant on file.

To mutate the current variant in place, match on the `Mut`'s `parts()`. It hands out the `Mut`s of the current variant's fields, in an enum named after yours, `PhoneNumberParts`, with the same variants:

```
use kladde::{Kladde, Durable};

#[derive(Durable)]
enum PhoneNumber {
    Mobile(String),
    Landline(String),
}

fn main() -> kladde::Result<()> {
    let mut number = Kladde::new(PhoneNumber::Mobile(String::from("555-01")));

    let mut number_mut = number.get_mut();
    match number_mut.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.
    number_mut.set(PhoneNumber::Landline(String::from("555-0123")))?;
    assert!(matches!(number.get(), PhoneNumber::Landline(digits) if digits == "555-0123"));
    Ok(())
}
```

Each field's `Mut` writes inside its field, so the variant stays what it was. `set` replaces the whole value, which is how the variant changes: it splices the new value's encoding over the old one's. `parts()` borrows the enum's `Mut`, so the compiler stops you from calling `set` while you still hold a field's `Mut`.

The discriminant is Rust's own: the one you write (`Mobile = 1`), or else one more than the previous variant's, counting from 0; negative discriminants and signed or pointer-sized `repr`s are not supported. The file holds it as a varint, so discriminants below 128 take one byte. Inside a `Fixed` value it takes a fixed width: the smallest of 1, 2, 4 or 8 bytes that holds the largest discriminant, unless an integer `#[repr]` such as `#[repr(u16)]` fixes it. Explicit values on variants with fields need a `#[repr]` anyway, as Rust requires:

```
#[derive(Durable)]
#[repr(u32)]
enum PhoneNumber {
    Mobile(String) = 1,
    Landline(String) = 2,
}
```

51.6 Trait objects

A `Box<dyn Shape>` is not durable: a file can hold only values whose types it can name. **Hold an enum of the shapes instead, and let it implement the trait by forwarding to its variants**: mark the trait `#[kladde::dispatch]`, and the enum `#[kladde(dispatch(Shape))]`.

```
use kladde::{Durable, Kladde};

#[kladde::dispatch]
trait Shape {
    fn area(&self) -> f32;
}

#[derive(Durable)]
struct Circle { r: f32 }
impl Shape for Circle {
    fn area(&self) -> f32 { 3.0 * self.r * self.r }
}

#[derive(Durable)]
struct Rect { w: f32, h: f32 }
impl Shape for Rect {
    fn area(&self) -> f32 { self.w * self.h }
}

#[derive(Durable)]
struct Frame { outer: Rect, hole: Rect, angle: f32, label: u32 }
impl Shape for Frame {
    fn area(&self) -> f32 { self.outer.area() - self.hole.area() }
}

#[derive(Durable)]
#[kladde(dispatch(Shape))]
#[repr(u32)]
enum AnyShape {
    Circle(Circle) = 1,
    Rect(Rect) = 2,
    Frame(Box<Frame>) = 3,  // boxed, so that a circle stays small in memory
}

fn main() -> kladde::Result<()> {
    let mut shape = Kladde::new(AnyShape::from(Circle { r: 1.0 }));
```

```
    assert_eq!(shape.get().area(), 3.0);

    let frame = Frame {
        outer: Rect { w: 3.0, h: 3.0 },
        hole: Rect { w: 1.0, h: 1.0 },
        angle: 0.0,
        label: 7,
    };
    shape.get_mut().set(frame.into())?;   // `From<Frame>` boxes it
    if let AnyShapeParts::Frame(mut f) = shape.get_mut().parts() {
        f.hole_mut().w_mut().set(2.0)?;
    }
    assert_eq!(shape.get().area(), 7.0);
    Ok(())
}
```

The enum implements every method of the trait, each forwarding to the value its current variant holds, and converts `From` each variant's value. It is an ordinary durable enum otherwise: you mutate a variant through `parts()`, and switch variants with `set`.

Each variant holds one value, whose type implements the trait, or the enum fails to compile, at the variant. The discriminants are optional, as on every enum, but explicit ones keep your files readable when you add or reorder implementations. **A variant may box its value**, which keeps one large implementation from making every value of the enum as large in memory; it changes nothing in the file, so you can box or unbox a variant later.

The trait's methods must take `self`, `&self` or `&mut self`, and mention `Self` nowhere else, and the trait and the enum must live in one crate; the derive macro says why.

51.7 Generics

Type parameters work, and each gets a `Durable` bound:

```
#[derive(Durable)]
struct Labelled<T> {
    label: String,
    value: T,
}
```

Lifetime and const parameters are not supported.

51.8 What the derive needs

Every persisted field must be `Durable`. In practice that means:

- primitives and `String` — fine as-is;
- `Vec<T>` — fine where `T` is fixed-size; for any other `T` it fails to compile, and the message points you to `DurableVec<T>`;
- kladde containers — fine as-is;
- your own derived types — fine, and they nest arbitrarily;
- `HashMap<K, V>`, `PathBuf` and other standard types — **not** `Durable`. Use the kladde equivalents, such as `DurableMap<K, V>`. This is deliberate: a field's type says whether it is durable, and the compile error tells you so.

51.9 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 encodes

only the fields you want persisted.

Doing so does **not** make your type opaque to the format. A hand-written implementation that encodes 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.

51.10 Encoding stability

A few things you should know before you ship a file format built on a derived type.

Field order is encoding. Reordering fields in the source changes the order of their encodings, 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 encoding only through implicit discriminants. A variant without an explicit discriminant takes the next number, so inserting or reordering variants renumbers 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 in a `Fixed` value is encoding 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 `Fixed` value of the type, 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 52

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."

52.1 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.

52.2 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 takes `String` and a `Vec` of fixed-size elements as they are, but needs its own container types for maps and for vectors of anything else.

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.

52.3 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 incremental 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.

52.4 They are not exclusive

`DurableBlob<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 serialized again in full on every change, so each edit costs time in proportion to the blob's size, the very cost you came to kladde to avoid, even though only the bytes that changed reach the file. Use it at the leaves, for things that change rarely.

52.5 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.

52.6 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 53

Durability and flushing

What is guaranteed when, and what a crash actually costs you.

53.1 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 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.

53.2 What a crash costs

After an application crash: nothing that returned `Ok`. If `push` returned `Ok`, 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 is one transaction, and so is each transaction you open, so recovery replays all of it or none of it: a push that splices in an element and rewrites the vector's count is never half there.

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.

53.3 Flushing

You do not normally have to think about this.

Flushing is **automatic**: a mutation that would not fit in what is left of the journal's budget folds the journal before it returns. 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.

An interactive application calls `flush_if_journal_exceeds(0.5)` from its idle callback. It flushes only if the journal holds at least that fraction of its budget, and returns whether it did, so it costs next to nothing when there is little to fold. Flushing while the user pauses keeps the automatic flush, whose `fsync` blocks the mutation that triggers it, out of the middle of a gesture such as a drag. Keep the fraction well above zero: a flush ends the journal, and with it every cut that is still waiting to be pasted, which is then encoded again when it is pasted.

```
if journal.flush_if_journal_exceeds(0.5)? {
    // folded while the user paused
}
```

`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.

53.4 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`.

53.5 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, Durable};

#[derive(Durable)]
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 both = tx.get_mut();
    let AccountsParts { mut checking, mut savings } = both.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.

53.6 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.get_mut().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 54

Implementing Durable by hand

Advanced. Most types should be derived or built from the containers. Reach for a hand-written implementation when neither fits.

54.1 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 encodes only the fields you want.

An encoding you control precisely — you want your type's bytes to be exactly so, rather than what the derive would produce.

A container — something with a non-trivial internal structure, which keeps an index of where its children start. 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 `DurableBlob<T>` instead, and to accept its costs.

54.2 The trait

```
pub trait Durable: Sized {
    const FIXED_SIZE: Option<usize>;

    fn encoded_len(&self) -> usize;
    fn encode(&self, out: &mut Encoder<'_>);
    fn decode(input: &mut Decoder<'_>) -> Result<Self, Error>;

    type Mut<'r, B: Backend + 'r>: Mut where Self: 'r;
    fn mutate<'r, B: Backend + 'r>(&'r mut self, region: Region<'r, B>) -> Self::Mut<'r, B>;

    fn describe_local(builder: &mut SchemaBuilder) -> TypeDescriptor;
}
```

A value of your type is plain, in-memory data; the trait says how it is encoded into the file's one sequence of bytes, and how a `Mut` mutates that encoding.

`FIXED_SIZE` is `Some(n)` if every value encodes to `n` bytes, and `None` if the size depends on the value. A fixed size lets a containing struct find the fields after yours at constant offsets, lets a vector of your type find its elements by multiplication and keep no index, and lets a plain `Vec` of your type be durable.

`encoded_len`, `encode` and `decode` are the encoding: how many bytes this value takes, writing them into an `Encoder`, and reading them back from a `Decoder`, which fails on bytes that no value encodes to. They never see where the value lies: the encoder appends to a buffer, and the decoder reads front to back.

`mutate` borrows the value together with its **region** — the substring of the sequence that holds its encoding — and returns the `Mut` your mutating methods live on. A region offers `write`, `splice`, `splice_value`, `move_bytes` and `paste`, all at offsets relative to the value's start and checked against its length, and `child` to make a region for a part of the value.

`describe_local` builds your type's descriptor, which must match the bytes.

54.3 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.

54.4 A complete example

A colour of three bytes, with a `Mut` that sets the whole colour or only its red channel.

```
use kladde::{
    Backend, Bounded, Decoder, Encoder, Error, Field, Mut, Durable, Region, SchemaBuilder,
    TypeDescriptor,
};

#[derive(Clone, Copy)]
pub struct Rgb {
    r: u8,
    g: u8,
    b: u8,
}

pub struct RgbMut<'r, B> {
    inner: &'r mut Rgb,
    region: Region<'r, B>,
}

impl<B: Backend> RgbMut<'_, B> {
    /// Replaces the whole colour: three bytes, written in place.
    pub fn set(&mut self, value: Rgb) -> Result<(), Error> {
        self.region.write(0, &[value.r, value.g, value.b])?;
        *self.inner = value;
        Ok(())
    }

    /// Replaces the red channel alone: one byte.
    pub fn set_red(&mut self, r: u8) -> Result<(), Error> {
        self.region.write(0, &[r])?;
        self.inner.r = r;
        Ok(())
    }
}
```

```
impl<B> Mut for RgbMut<'_, B> {
    type Durable = Rgb;
    fn as_durable(&self) -> &Rgb {
        self.inner
    }
}

impl<B> std::ops::Deref for RgbMut<'_, B> {
    type Target = Rgb;
    fn deref(&self) -> &Rgb {
        self.inner
    }
}

impl Durable for Rgb {
    const FIXED_SIZE: Option<usize> = Some(3);

    fn encoded_len(&self) -> usize {
        3
    }

    fn encode(&self, out: &mut Encoder<'_>) {
        out.bytes(&[self.r, self.g, self.b]);
    }

    fn decode(input: &mut Decoder<'_>) -> Result<Self, Error> {
        let [r, g, b] = input.array()?;
        Ok(Rgb { r, g, b })
    }

    type Mut<'r, B: Backend + 'r> = RgbMut<'r, B>;

    fn mutate<'r, B: Backend + 'r>(&'r mut self, region: Region<'r, B>) -> RgbMut<'r, B> {
        RgbMut { inner: self, region }
    }

    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 Durable>::describe(builder),
                })
                .collect(),
        }
    }
}

/// The fixed form, which `Fixed<Rgb>` holds: the same three bytes.
impl Bounded for Rgb {
    const FIXED_FORM_SIZE: usize = 3;

    fn encode_fixed(&self, out: &mut Encoder<'_>) {
        self.encode(out);
    }
```

```
    fn decode_fixed(input: &mut Decoder<'_>) -> Result<Self, Error> {
        Self::decode(input)
    }
}
```

Four things worth noting.

`FIXED_SIZE` and `encoded_len` say 3 because `encode` writes exactly three bytes. Every offset computed by a containing struct depends on that number being right.

The `Mut` is a separate type, generated for you by the derive macro but written out here. It holds the value and its region, 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. `set_red` writes one byte at offset 0 of the region, which is the red channel's place in the encoding; the `Mut` knows nothing about where the colour lies in the file.

`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 encoding genuinely is not decomposable; see Your descriptor.

`Bounded` is optional: it gives the type a fixed form, so that `Fixed<Rgb>` exists. A fixed-size type's fixed form is usually its encoding byte for byte, as here; for a type with varints inside, it lays every integer out at its full width.

54.5 A value whose size changes

If your type's encoding can change size, its whole-value `set` splices instead of writing: `region.splice(0, region.len(), &new.to_bytes())`, which records a write where the sizes happen to agree. `kladde::replace(&mut value, new, &region)` does exactly that and then updates the value. For a large value that is replaced whole but changes little, `kladde::splice_changed(&region, at, old, new)` records only the range between the common prefix and suffix of the old and the new bytes, as `String`'s `set` and `DurableBlob` do. The region reports the change of size to the regions around it, which update their indexes, so nothing else needs doing.

A type with several parts, some of whose sizes change, hands each part a region of its own with `region.child(at, len, node, index)`, where `node` is the type's index of where its parts start, which the child's region updates when the child changes size; the vector of `kladde-types` is the example to read. A mutation that changes two places — a container's element and its count, say — opens `region.transaction(|r| ...)`, so that both reach the journal together.

54.6 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 encode**, 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 encoding genuinely is not decomposable into fields, variants and sequences — an externally serialized payload, say. "Hand-written" and "opaque" are different axes, and conflating them makes your type needlessly invisible to tooling.

54.7 The obligations you are taking on

A hand-written implementation is trusted, not checked. You are promising:

- `FIXED_SIZE` and `encoded_len` match what `encode` writes;
- `encode` and `decode` are exact inverses, and the encoding is the canonical one for your descriptor;
- every `Mut` method leaves its region holding the value's encoding when its transaction ends;
- your declared descriptor matches your actual bytes.

The compiler enforces none of these, but the third is easy to test: flush, and compare the file's sequence with `to_bytes()` of the value in memory.

Chapter 55

Crate layout

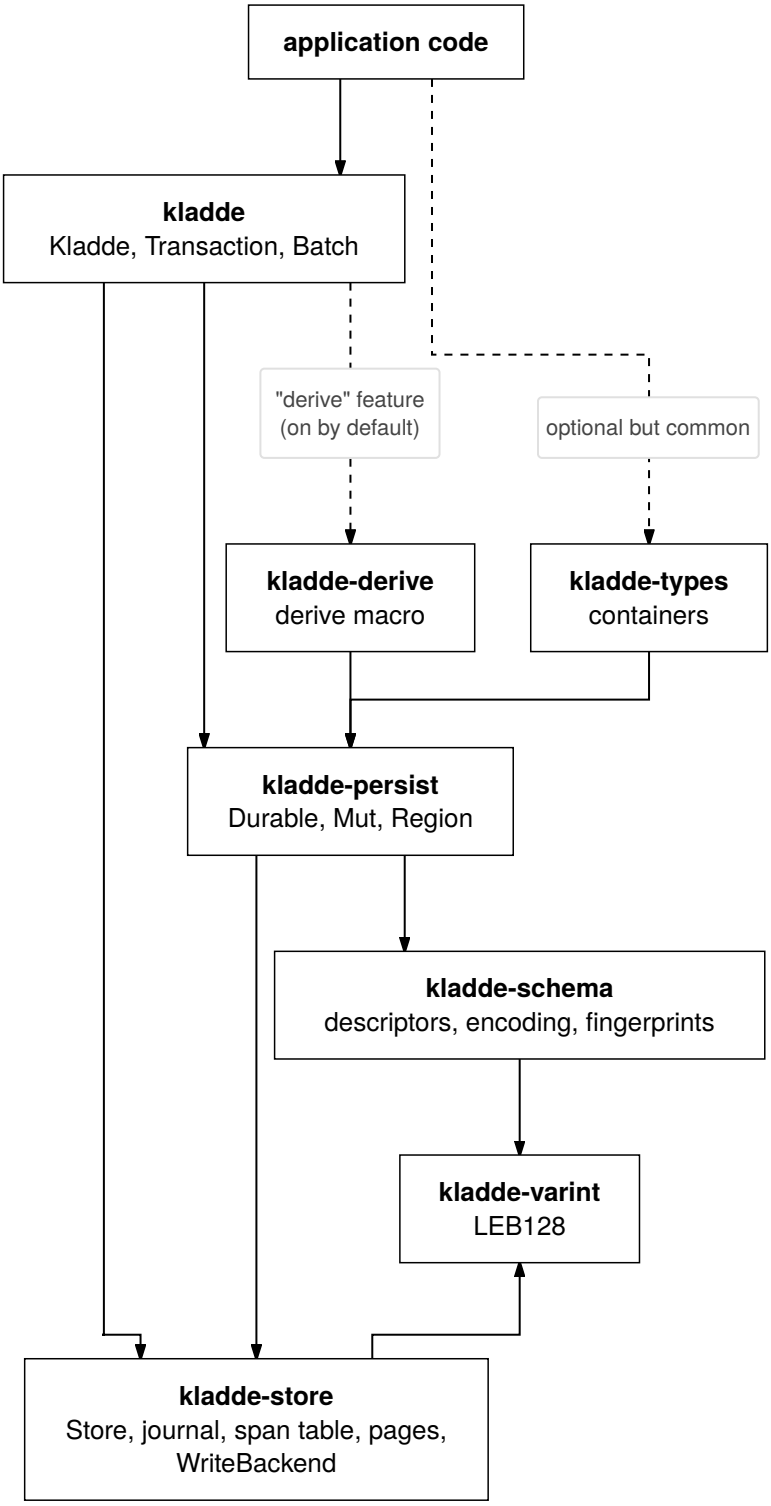
The workspace, what depends on what, and where the language-neutral half ends.

55.1 The crates

crate	role
`kladde-store`	the storage layer: the file, pages, the span table, the journal, the flush, and consolidation, behind the `WriteBackend` trait. Type-agnostic; knows nothing about serialization.
`kladde-persist`	the serialization layer: `Durable`, `Bounded`, `Mut`, `Region`, `Encoder` and `Decoder`, the scalars, `String`, `Vec` of fixed-size elements, tuples, `Option`, `Box` and `Fixed`, cuts, and the schema binding.
`kladde-schema`	type descriptors, their canonical encoding, fingerprints, and the rules of nesting. Depends on nothing but `kladde-varint`.
`kladde-varint`	LEB128 varints and zigzag.
`kladde-types`	the built-in containers that keep an index or a serialization in memory: `DurableVec`, `DurableMap`, `DurableBlob`. The default collection, not a layer — it uses only the public API any library could.
`kladde-derive`	the `#[derive(Durable)]` macro, and the `#[kladde::dispatch]` attribute with which a derived enum stands in for a trait object. Reached through `kladde`'s `derive` 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 derive macro and everything a `Durable` impl names, so an application needs this crate and nothing below it.

Two more crates sit beside them, used by the measurements and nothing else: `kladde-svg`, an SVG drawing held as a durable value, with `svg-bench` among its binaries, and `kladde-bench`, with `text-bench`; together they drive the evaluation. A third, `kladde-inspect`, is a command-line tool that reports where the bytes of any kladde file go, from `kladde-store`'s `inspect` module and the file's own schema.

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 one byte sequence, independently of anything above it.


application code
kladde
Kladde, Transaction, Batch
"derive" feature
(on by default)
optional but common
kladde-derive
derive macro
kladde-types
containers
kladde-persist
Durable, Mut, Region
kladde-schema
descriptors, encoding, fingerprints
kladde-varint
LEB128
kladde-store
Store, journal, span table, pages,
WriteBackend

55.2 The parts

55.2.1 `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.

55.2.2 `kladde-persist` — the serialization layer

Where types meet bytes. `Durable` and `Bounded`, the `Mut` counterpart, `Region` and the `Node` a container keeps for its children, the `Encoder` and `Decoder`, `Cut`, and the implementations for scalars, `String`, `Vec` of fixed-size elements, tuples, `Option`, `Box` and `Fixed`. Knows about the store's `WriteBackend`; knows nothing about any particular container or derived type.

55.2.3 `kladde-schema` — type descriptors

The Rust implementation of the language-independent schema model. It depends on neither the store nor `Durable`, 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.

55.2.4 `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.

55.2.5 `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 `Mut`s, transactions, and batches.

55.3 The seam

For anyone porting kladde, one boundary matters more than the rest.

Below `kladde-persist`, the design is language-neutral. The span table, the journal, the flush, the consolidation policy, the schema model, and the encoding of values all translate more or less directly; they are documented in Specification and Implementation rather than here, and so are the regions and the containers' indexes that Values and their positions describes.

At and above `kladde-persist`, the design is Rust-specific and should be redone.

- `Durable`/`Mut` exists because Rust cannot intercept a field assignment.
- `WriteBackend` takes `&self` because sibling `Mut`s share one backend by reborrow, which Rust's borrow checker otherwise forbids.
- `FIXED_SIZE` as an associated constant exists because Rust can compute field offsets at compile time. A language without compile-time constant folding will compute them some other way — probably once per type at startup, which is fine, but it changes the shape of the code.
- `Bounded` is a subtrait so that `Fixed<T>` fails to compile for a `T` without a fixed form; another language checks it however its type system allows.

A Python port should keep the storage layer almost verbatim and throw away the `Mut`s entirely, intercepting assignments with property hooks. A C++ one might use proxy objects with `operator=`; a Java one might use bytecode enhancement, which is what db4o did.

55.4 Cross-cutting concerns

Crash consistency is a property of the journal and the flush, and of the one rule a container must keep: every edit it makes, such as an element and its count, goes into one transaction (ordering); the `Muts` wrap every mutation in one.

Schema evolution touches the load path, the mutation path, and the flush, because a value read in a foreign schema must not then be written in the native one unless the file is upgraded; see the specification. Until it exists, a file whose root fingerprint differs from the application's is refused at open.

Chapter 56

Durable and Mut

The central traits, why mutation needs a `Mut`, and why it returns a `Result`.

A data type encodes, decodes, and mutates its region; it never allocates, frees, or knows a position.

56.1 `Durable`

```rust
pub trait Durable: Sized {
    /// `Some(n)` if every value encodes to `n` bytes.
    const FIXED_SIZE: Option<usize>;

    fn encoded_len(&self) -> usize;
    fn encode(&self, out: &mut Encoder<'_>);
    fn decode(input: &mut Decoder<'_>) -> Result<Self, Error>;

    type Mut<'r, B: Backend + 'r>: Mut where Self: 'r;
    fn mutate<'r, B: Backend + 'r>(&'r mut self, region: Region<'r, B>) -> Self::Mut<'r, B>;

    fn describe_local(builder: &mut SchemaBuilder) -> TypeDescriptor;
}

/// A type with a fixed form, which `Fixed<Self>` holds.
#[diagnostic::on_unimplemented(message = "`{Self}` has no fixed form", /* ... */)]
pub trait Bounded: Durable {
    const FIXED_FORM_SIZE: usize;
    fn encode_fixed(&self, out: &mut Encoder<'_>);
    fn decode_fixed(input: &mut Decoder<'_>) -> Result<Self, Error>;
}

pub struct Fixed<T: Bounded>(pub T);
```

A `Durable` 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. Besides `describe_local`, the trait offers `to_bytes`, `from_bytes`, `describe`, `schema` and `fingerprint`, with defaults.

56.1.1 `FIXED_SIZE`

`Some(n)` if every value encodes to `n` bytes, as the specification has it, and `None` otherwise. A `DurableVec` decides by it whether to keep an index of where its elements start, a plain `Vec<T>` is durable only where it is `Some` (a const assertion refuses any other `T` at compile time), and the derive decides by it whether a field after another has a constant offset. It stays an `Option` on `Durable`, rather than a subtrait, because a vector branches on it at compile time, which a subtrait could support only with specialization.

A `Box<T>`'s `FIXED_SIZE` is `None` whatever `T`'s: a type that holds itself through a box would otherwise ask its own size of itself while the compiler evaluates it, which rustc refuses as a cycle (E0391) rather than reading it as "not fixed-size". The derive cannot tell a recursive type from the tokens of one type, and a box is the one indirection among kladde's types that could pass a fixed size on, so answering `None` there cuts every cycle; a vector of boxes then keeps an index it could do without, and nothing on file changes.

56.1.2 `encode` and `decode`

`encode` appends the value's encoding to an `Encoder`, and `decode` reads it from a `Decoder`; they differ because writing goes into one buffer, while reading crosses spans. An `Encoder` appends to a buffer it borrows and offers varints, zigzag, and UTF-8 with its terminator; a value is encoded once per record, into the value's own buffer, which the store then copies into the record — a copy a store method taking an encoding closure would save. A `Decoder` reads the sequence from the mirror through the rope, across span boundaries, without copying, refuses an encoding that is not canonical, and says where a decode failed. `encoded_len` takes constant time for every type of the workspace: containers cache their total.

56.1.3 `Bounded` and `Fixed<T>`

`Fixed<T>` holds `T`'s values in their fixed form, which a subtrait, `Bounded`, states, so that no method panics for a type without one, and `Fixed<String>` fails to compile where it is written, with a message that names the fix. The scalars, `Fixed`, and the `Options`, `Boxes` and tuples of `Bounded` types are `Bounded`, and a derived type is if it is marked `#[kladde(bounded)]` (the derive). A `Box<T>` takes `T`'s fixed form, as it takes `T`'s encoding and descriptor, so boxing a field keeps `Fixed` of what holds it. Its conditional impl never meets the cycle that its `FIXED_SIZE` cuts, since a recursive type is never `Bounded`; a recursive type marked `#[kladde(bounded)]` fails to compile, though with rustc's cycle error (E0391) rather than kladde's message. A type with a fixed size and no fixed form is possible, and harmless, since nothing can wrap it in `Fixed`.

`FIXED_FORM_SIZE` is the bound on a value's size that `Fixed<T>` needs, so `Durable` declares no other maximum: `Fixed<T>` stores the fixed form, every integer at its full width, rather than `T`'s encoding padded to its longest, in which a field would move whenever a varint before it grew. A fixed-size type's fixed form is its encoding byte for byte, except where an enum's integer `#[repr]` sets a discriminant wider than the one byte its encoding takes: `#[repr(u32)] enum Pinned { A, B }` has `FIXED_SIZE` 1 and `FIXED_FORM_SIZE` 4.

The `Mut` of a `Fixed<T>` replaces the value whole, an overwrite of its fixed size, since a `Mut` of one of `T`'s fields would have to write that field in its fixed form, which would take a second `Mut` for every type. That is visible to the caller: the `Mut` offers `set(Fixed<T>)` and reading, and no `Mut` of a field of `T`.

56.2 Regions

```
pub trait Backend: WriteBackend + Copy {}   // any `WriteBackend` that is `Copy`, such as `&Store`
pub struct Region<'r, B> { /* the backend, and where the value starts and how long it is */ }
```

A region is the value's substring of the sequence, as the reference algorithms describe it: `len()`, `write(at, bytes)`, `splice(at, removed, bytes)`, `splice_value(at, removed, &value)`, `move_bytes(from, len, to)`, `paste(at, removed, cut, bytes)`, `transaction(f)`, and `child(at, len, node, index)`. `paste` takes the value's encoding along with the cut, and writes it only if a flush has folded the journal that made the cut. `Backend` is any `WriteBackend` that is `Copy`, so that a `Mut` hands its backend to its children and to a cut by value.

A region's start is one of three things: the root's, an offset from an anchor region that is constant while the region lives, or *indexed* — looked up in its parent's `Node` at every write, for a region made together with its siblings. When a region changes size, it tells the `Node` it was made from, which updates where the children behind it start, and its parent's region, which tells its own parent in turn; lengths and indexes sit in `Cells`, so that regions made together, which share their parent by shared reference, can report through it.

`transaction(f)` poisons the store if `f` fails, since the regions around may have heard of a size change by then; a `Mut` checks what can fail before it opens one.

A child made on its own borrows its parent region mutably, so the parent cannot write while the child exists, and no sibling can move it.

56.3 `Mut`

```
pub trait Mut {
    type Durable;
    fn as_durable(&self) -> &Self::Durable;
}
```

A `Mut` borrows the value and its region for a lifetime, and stands where Rust's `&mut T` would: `Durable::mutate` hands it out as `get_mut` hands out a `&mut T`, and `Kladde::get_mut` is its entry point. It is an RAII token proving exclusive, recording access. A `Mut`'s `Durable` need not be `Self`, since a `Box<T>` uses `T`'s `Mut`.

Every mutating method on a `Mut` does two things: it records the mutation through its region, 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; `Mut`s implement `Deref`, and not `DerefMut`, which would let `point_mut.x = value` compile and persist nothing. Scalars, and anything whose `Mut` offers only a whole-value `set`, use the generic `SetMut`.

56.3.1 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 records nothing; a mutation of several records is one transaction, appended whole or not at all.

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`.

56.3.2 `Mut`s are generated per type

Not one shared generic wrapper, and the reason is Rust's orphan rules. A single `Mut<T, B>` struct 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, regardless of what their parameters are filled with. So a library author could not write ordinary methods on their own `Mut`. Per-type generation means `MyTypeMut` is local to the crate that declared `MyType`, and ordinary `impl` blocks just work.

56.4 Why an automatic flush cannot tear a mutation

The store flushes by itself whenever a mutation would not fit in what is left of its journal's budget, and it does so from inside the mutating call that closed it — while a `Mut` is alive. That is safe for two reasons, one per layer.

The flush touches no `Durable` value and moves no position. It folds journal records into pages; it never reads or writes an in-memory value, and positions in the sequence, which `Mut`s hold, mean the same after it.

A flush lands only on a transaction boundary. It runs between appends, and every append is a whole transaction, so every state a flush commits is a valid sequence.

56.5 The chain, end to end

```
notes.get_mut()            // NotesMut  — the root's region: the whole sequence
     .lines_mut()          // DurableVecMut — a child region, after the title
     .push(value)?;        // splices the element and the count, then mutates memory
```

Each `_mut()` makes a child region from its parent's, at the field's offset, and reborrows the same backend. The caller supplies a backend exactly once, at the outermost `Mut`.

56.6 What this costs

- **Verbosity.** `x.get_mut().field_mut().set(v)?` where other languages write `x.field = v`.
- **Borrow contention.** A `Mut` borrows the root, so nothing else can touch the structure while it lives.
- **Partial borrows only through `parts()`.** `Mut`s of two fields at once come from `parts()`, which looks each field's start up at every write; there is no `split_at_mut` for a vector yet.

Chapter 57

Containers

How kladde-rs realises the built-in containers, and the choices that are Rust's.

Text and vectors of fixed-size elements are the standard `String` and `Vec<T>`, durable in kladde-persist, since they need nothing in memory beyond what the standard types hold. `DurableVec<T>`, `DurableMap<K, V>` and `DurableBlob<T>` live in kladde-types, since each keeps an index or a serialization of its own in memory. All of them are **hand-implemented** against `Durable`, `Mut` and `Region`, not generated. Everything they use is public, so a third-party library of durable types can do anything they do.

57.1 `Vec<T>` and `DurableVec<T>`

`Vec<T>` is durable when `T` is fixed-size, and `DurableVec<T>` for every `T`; `DurableVec<T>` is the type for elements whose size depends on their value. Both have the same encoding and the same schema, a Sequence: the count as a varint, then the elements.

`Vec<T>` finds element `i` at `i` times the element's size after the count, so the standard vector holds everything it needs. A `T` whose `FIXED_SIZE` is `None` is refused at compile time by a const assertion in `Vec<T>`'s `Durable` implementation, whose message names `DurableVec<T>`. Its `Mut` is `VecMut`, with `set`, `insert`, `push`, `remove`, `pop`, `clear` and `get_mut`; every insertion and removal is a splice of the element and a rewrite of the count, in one transaction.

`DurableVec<T>` keeps, in memory, a `Vec<T>`, and for a `T` whose `FIXED_SIZE` is `None`, a Fenwick tree of the elements' encoded sizes in 32-bit cells, behind a cached total; for a fixed-size `T`, element `i` starts at `i` times its size after the count, and it keeps no tree. It adds to `Vec<T>`'s methods those that a `Vec` has no room for: `delete`, `move_element`, and cuts.

The tree appends and pops in `O(log m)`, answers where element `i` starts in `O(log m)`, and is built again in `O(m)` after an insertion, a removal or a reordering elsewhere, beside the `O(m)` move of the `Vec`'s elements. It lies in cells and implements `Node`, so the element `Mut`s that `parts`-like accessors make together, or a future `split_at_mut`, can look their starts up through it while sharing it.

- **`push` and `insert`** splice the element in with `splice_value` and rewrite the count, in one transaction.
- **`pop`, `remove` and `delete`** splice it out and rewrite the count; `remove` and `pop` return the value, and `delete` drops it.
- **`move_element(from, to)`** records one `Move`, which writes no element's bytes.
- **`get_mut(i)`** makes the element's region, at the start the tree gives, and the element's `Mut`; a change of the element's size reaches the tree through the region.
- **`set`** replaces the whole vector.
- **`cut(i)`, `insert_cut` and `push_cut`** move an element between vectors as a cut and a paste, below.

`T` needs only `Durable` — no `Clone`, no `Serialize` — since a `Mut` method encodes a value into the record while only borrowing it. The vector reads through `Deref<Target = [T]>`, so a slice's whole read API works on it.

The tree's field takes 40 bytes even where it is empty or the elements are fixed-size: 24 for its `Vec` and 16 for its cached total and the count's varint length, which it adds when it answers for `Muts` made together. Neither number needs a field — the total can live in the tree's allocation, and the count's length is the vector's to pass in — which would leave the 24 bytes of an empty `Vec`, or 8 with a thin one.

57.2 `String`

Encoding: Text: UTF-8, then `0xFF`. **In memory:** the `String` itself.

Its `Mut` is `StringMut`, whose `replace_range`, `push_str`, `insert_str`, `truncate` and `set` are each at most one splice, since text has no count to rewrite. **`set` records only what changed**: one splice of the bytes between the longest prefix and the longest suffix that the old and the new text share, or nothing if they are equal. So code that sets a whole text at every change, such as a path's data rewritten at every step of a drag, records a few bytes rather than the whole text, at the cost of one pass over both texts. The comparison is public as `kladde_persist::splice_changed`, for hand-written `Muts` of other values that are replaced whole.

Text is UTF-8; a byte buffer is a `Vec<u8>`.

57.3 `DurableMap<K, V>`

Encoding: a Map, a descriptor kind of its own with a key type and a value type: the count as a varint, then each entry's key and value, in the map's order. **In memory:** an index map — the entries in a `Vec<(K, V)>`, in their order on file, and a hash table of their indices, its own, with open addressing over 32-bit positions, probed linearly and at most half full — and for variable-size entries a Fenwick tree of their sizes.

- **`insert`** appends a new key, or splices a new value over an existing key's, returning the old one.
- **`remove` and `delete`** swap: the last entry moves into the removed one's place by a `Move`, the removed entry is spliced out, and the count goes down, all in one transaction; only the moved entry's index changes in memory.
- **`get_mut`** hands out a `Mut` for a value, never for a key: keys are immutable.

Each key is stored once, in the entries, so `K: Clone` is not required.

57.4 `DurableBlob<T>`

Encoding: an Opaque without a fixed size: the serde payload, prefixed with its length. Wraps any `T: Serialize + DeserializeOwned + Default`.

The escape hatch for foreign types that can get neither their own wrapper nor a derive, and deliberately the least attractive option: serialized again in full on every change, opaque to tooling, and the only thing in the workspace that needs a serialization dependency — hence the feature gate. Its in-place edit is a closure, `update(|value| ...)`, rather than a handle that persists when dropped, since `Drop` cannot report an error.

`set` and `update` record only the bytes of the serialization that changed. The blob keeps its last serialization, compares the new one with it, and splices in the range between their common prefix and suffix with `splice_changed`; where the length changes, the rewritten length prefix goes in the same transaction. Editing one nested field of a large blob then writes little, though how little depends on the serialization: postcard's varints and length prefixes put a changed byte near the front of everything behind a value that changes size, so the one range that holds every change can reach far.

57.5 Cuts

A container's `cut(i)` removes element `i` as `remove(i)` does, and returns a `Cut<T, B>`: the in-memory value, and the store's handle on the removed bytes. Cutting is an ordinary removal, recorded in a transaction of its own; it holds no transaction open.

A cut is weak: it lives as long as the journal that made it. A paste while that journal lasts writes none of the value's bytes, since it records a `Paste`; a paste after a flush has folded that journal encodes the value again, as an ordinary splice would. Either way the sequence ends up the same, so a `Cut` may be held across transactions and flushes, and `Cut::is_alive()` tells whether a paste would still write none of its bytes.

- `push_cut` and `insert_cut` take a `Cut<T, B>`, record the paste, and move the value into memory. They are methods of their own, rather than `push` taking either a value or a cut, since one method taking either would need a conversion trait whose two impls Rust's coherence rules make fragile.
- A `Cut` dereferences to its value but gives no mutable access, since its bytes must stay the value's encoding until they are pasted.
- It is neither `Copy` nor `Clone`, and typed by its value's type, so it is pasted at most once, and a `Cut<Fixed<Shape>>` cannot go where a `Shape` goes.
- It borrows the store, `B = &Store`, not the `Mut` it came from, so the application can drop the source's `Mut` and make the destination's while it holds the cut.
- Dropping it unpasted drops the value, which the container removed when it cut it; nothing is recorded, so a drop has no failure to report.

kladde-rs implements cuts on `DurableVec` only; `Vec`, the map, fields, `Option` and enums neither make nor take them yet, though any `Mut` method that splices a value out could return one.

57.6 What is missing

A rope, for large text with cheap lookups by position in memory. The file already takes splices in long text at constant cost; what a `String` costs is the `O(n)` shift of its in-memory bytes.

`split_at_mut` on the vector, for `Mut`s of two runs of elements at once. The tree already lies in cells and answers any element's start, so it would serve as the two runs' index as it is; it was left out only because nothing measured needs it.

Chapter 58

The derive macro

What `#[derive(Durable)]` generates, and why.

58.1 What it generates

For a struct or enum, four things:

1. a `Durable` implementation — `FIXED_SIZE`, `encoded_len`, `encode`, `decode`, and the schema descriptor;
2. a **`Mut` type**, `MyTypeMut`, with a `_mut()` accessor per field, a `parts()` method handing out a `Mut` for every field at once, and a whole-value `set`;
3. a `Deref` implementation on the `Mut`, so read-only methods stay available — and no `DerefMut`, which would hand out a `&mut` whose changes no journal record describes;
4. with `#[kladde(bounded)]`, a `Bounded` implementation: the fixed form.

```
#[derive(Durable)]
struct Point { x: f32, label: String, n: u32 }
```

generates, in essence:

```
impl Durable for Point {
    const FIXED_SIZE: Option<usize> = sum_sizes(&[f32::FIXED_SIZE, String::FIXED_SIZE, u32::FIXED_SIZE
    fn encoded_len(&self) -> usize { self.x.encoded_len() + self.label.encoded_len() + self.n.encoded_
    fn encode(&self, out: &mut Encoder<'_>) { self.x.encode(out); self.label.encode(out); self.n.encod
    fn decode(input: &mut Decoder<'_>) -> Result<Self, Error> {
        Ok(Point { x: f32::decode(input)?, label: String::decode(input)?, n: u32::decode(input)? })
    }
    // ...
}

impl<'r, B: Backend> PointMut<'r, B> {
    fn label_mut(&mut self) -> StringMut<'_, B> {
        let offset = 4;                                  // after `x`, whose size is fixed
        let len = self.inner.label.encoded_len();
        self.inner.label.mutate(self.region.child(offset, len, None, 1))
    }
    fn n_mut(&mut self) -> SetMut<'_, u32, B> {
        let offset = 4 + self.inner.label.encoded_len(); // after a field whose size varies
        let len = self.inner.n.encoded_len();
        self.inner.n.mutate(self.region.child(offset, len, None, 2))
    }
}
```

A field's accessor makes a child region at the sum of the sizes of the fields before it: a constant up to

the first field whose `FIXED_SIZE` is `None`, folded by the compiler, and after that the sum of the earlier fields' encoded sizes, which the in-memory values give in constant time. The child region reports a change of the field's size to the struct's region, which passes it on up.

`parts()` makes every field's `Mut` at once, from a `FieldSizes` node it fills with the fields' sizes: a field after the first variable-size one is *indexed*, and looks its start up in the node at every write, so that a sibling before it may change size meanwhile (handles made together).

58.2 The aggregate records nothing of its own

A struct is its fields' encodings back to back, so its `Mut` has nothing of its own to write beyond `set`. Every accessor just makes a child region and the field's own `Mut`, recursively. This is why the macro is small: it computes offsets and forwards; all the actual work is in the leaf types.

58.3 Fixed forms

`#[kladde(bounded)]` implements `Bounded`, the fixed form that `Fixed<T>` holds: every field's fixed form, back to back, and for an enum the discriminant at a fixed width and its variant padded with zeros to the largest. It requires every field's type to be `Bounded`, failing at the type's definition otherwise, with the message `#[diagnostic::on_unimplemented]` gives, and bounds each type parameter by `Bounded`, so that a generic type such as `Tagged<T>` is `Bounded` where its parameter is.

The derive cannot implement `Bounded` unasked, exactly where the fields' types allow it: for a type without parameters, rustc refuses an impl whose where-clause names a concrete type that lacks the trait (E0277) rather than dropping the impl, short of the unstable `trivial_bounds`. So the author opts in. Opting in marks only the types that `Fixed` wraps and those inside them, none in kladde-svg; opting out would mark every type that holds text, a container or itself, 6 of kladde-svg's 15 derived types.

58.4 Primitives are `Durable` 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* `Durable`, because `impl Durable for i32` is `impl ForeignTrait for ForeignType` from anywhere else, which the orphan rules forbid; the same reasoning forces the scalar `Mut` types into that crate. The same holds for `String` and `Vec<T>`, whose `Durable` implementations live in kladde-persist beside the scalars'.

58.5 Enums

An enum is its discriminant as a varint, followed by the selected variant's fields, each variant encoded like a struct in its own right. It is fixed-size if every discriminant is below 128 and every variant takes the same fixed size. The discriminant is Rust's own: the explicit value where the author wrote one, otherwise one more than the previous variant's.

In the schema, every variant holds exactly one value, as the Enum descriptor has it, so the derive picks the value's type by the variant's shape:

- a unit variant holds `()`, the empty struct;
- a variant with one unnamed field holds that field's type;
- any other variant holds an anonymous struct of its fields, named after the variant, its positional fields named by position — `"0"`, `"1"`.

So `enum E { R { w: u32, h: u32 } }` and `enum E { R(Rect) }` with `struct Rect { w: u32, h: u32 }` have the same fingerprint and the same bytes, and moving a variant's fields into a struct of their own changes nothing on file. `Option`'s `None` holds `()` too.

The discriminant's width in the fixed form is as wide as its values need, unless an integer `#[repr]` fixes it. `#[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 `usize` has no fixed width. Without a `repr`, the width is a cliff: a 257th variant, or a discriminant of 256 or more, widens every `Fixed` value of the type; a `repr` keeps the fixed form's size when variants are added.

The `Mut`'s `parts()` hands out the `Mut`s 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 shape_mut.parts() {
    ShapeParts::Circle(mut radius) => radius.set(2)?,
    ShapeParts::Rectangle { mut width, .. } => width.set(3)?,
    ShapeParts::Origin => {}
}
```

Each field's `Mut` is based at the field's offset past the discriminant's varint and writes inside its field only, so the discriminant cannot change underneath it. And `parts()` borrows the enum's `Mut`, so `set` cannot switch the variant while a field's `Mut` is alive. `set` is how the variant changes: it splices the new value's encoding over the old one's. `parts()` on a temporary `Mut` works in a `match` but not in a `let … else`, which drops the temporary at the end of the statement; a consuming `into_parts(self)` would make the `let` form work too, and is proposed rather than built. An enum without any fields gets no `parts()`, since it has nothing to mutate in place.

58.6 Trait objects, as an enum

A file can hold only values whose types it can name, so where an application would hold a `Box<dyn Shape>`, it holds an enum of the shapes, which implements `Shape` by forwarding every method to its current variant. The trait is marked `#[kladde::dispatch]`, and the enum `#[kladde(dispatch(Shape))]` beside `#[derive(Durable)]`; every variant holds one value, of a type that implements the trait. The derive also implements `From` for every variant's type, and for the `T` of a variant that holds a `Box<T>`, so `AnyShape::from(Circle { .. })` builds the right variant.

On file the enum is an ordinary enum, with a variant per implementation, so the schema needs nothing new, and a C++ or Python implementation reads it as it reads any enum. A variant may hold its value in a `Box`, to keep a large variant from making every value of the enum large in memory: a box is persisted as what it holds, its `Mut` is the `Mut` of what it holds, and it is `Bounded` where what it holds is, so boxing or unboxing a variant changes neither the file nor the fingerprint.

Why an enum rather than `Box<dyn Trait>` *itself.* Rust cannot list a trait's implementations, so the variants have to be listed somewhere, and an enum is the list Rust already checks: two variants with the same explicit discriminant fail to compile (E0081), and a variant's type that does not implement the trait fails at the enum. A reader has to know every variant to decode a file, so the openness of `dyn` would buy nothing on file; collecting implementations from several crates at link time would make the fingerprint depend on which crates a binary linked. And `Durable` cannot be used through `dyn` — it requires `Sized`, has associated constants and a generic `Mut`, and `decode` returns `Self` — so a durable `Box<dyn Shape>` would need a helper trait, a table of decoders, and `Mut`s reached by downcasting, where the enum gets `parts()` and an exhaustive `match`. The enum is also faster, with no vtable and no allocation per value.

How the two macros meet. A derive sees only the enum, never the trait's methods, so `#[kladde::dispatch]` emits, beside the unchanged trait, a hidden `macro_rules!` macro, `__kladde_dispatch_Shape`, that knows the methods and writes the forwarding impl for an enum it is handed; it re-exports it next to the trait with `pub(crate) use`. The derive invokes it by the trait's path, with the trait's module path and the macro's name in place of the trait's, so the enum may be in any module, before or after the trait. For each variant it passes the pattern binding the held value, the value's type, and how to reach the value by reference, by mutable reference and by value, through the box for a variant that holds one; and the impl's `where` clause, with a bound of the trait on each variant's type, spanned at that type, so that a missing implementation is reported at the enum, and a generic enum gets the bounds it needs. This is the pattern of the `ambassador` crate; `enum_dispatch` instead passes the trait's tokens through global state between the two macros, which depends on the order rustc expands them in.

What it forwards. Every method of the trait, defaulted ones included, so that a variant's own implemen-

tation wins: methods taking `self`, `&self` or `&mut self`, generic ones, `async` ones, `unsafe` ones, and ones with `impl Trait` arguments. It refuses, at the trait, what it cannot forward: a method without a receiver, which no enum could choose a variant for; one that takes `Self` other than as its receiver, or returns `Self` or `impl Trait`, since each variant would bring or return a type of its own; and associated types and constants, which each variant may choose differently, and generic traits. A variant's type that two variants hold gets no `From`, nor does one that mentions a type parameter, since `From<T>` would overlap with every other variant's.

Limitations. The trait and the enum must be in one crate: a `macro_rules!` macro can be public only through `#[macro_export]`, which puts it at the crate root, where a macro-expanded one may not be named by path from its own crate. A box is recognized by its spelling, `Box<T>`, `std::boxed::Box<T>` or `alloc::boxed::Box<T>`, so a variant holding a box behind a type alias forwards to the box itself, which then has to implement the trait.

58.7 Generics and transparency

Type parameters are supported: each gets a `Durable` 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 `Mut` one `get_mut()` returning the field's `Mut`, as `#[serde(transparent)]` does.

58.8 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 `Durable` 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(Durable)]
#[kladde(crate = "kladde_persist")]
struct Node { /* ... */ }
```

58.9 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. Two more, modeled on `serde`'s, would serve types the derive cannot handle today:

- **`#[kladde(with = "path")]` on a field** of a type that is not `Durable`, a foreign one or one wanting an encoding of its own, delegating its encoding, descriptor and `Mut` to a module of functions, so that a foreign type can be encoded inline with a real descriptor rather than wrapped in a `DurableBlob`.
- **`#[kladde(bound = "...")]` on the type**, overriding the `Durable` bound the derive puts on every type parameter, for a parameter used only in `PhantomData`, or only inside a wrapper that needs a different bound.

Const generic parameters, which the derive rejects today.

Conveniences on `Muts`: `set_<field>(value)` on a struct's `Mut`, sparing the step through the field's accessor, and `field_{i}_mut()` on a tuple's `Mut`, which offers only `parts()` and `set`, as tuple structs' `Muts` offer it. A tuple's accessors cannot be generated by `macro_rules!` without a crate such as `paste` to build their names.

58.10 Testing compile failures

The macro's *compile errors* are part of its contract — a field of a type that is not `Durable`, such as a `PathBuf`, 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 59

Transactions

The Rust surface over the transaction and batch machinery: RAII types, and what happens when a panic unwinds through one.

59.1 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 `get_mut()`, 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.

59.2 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.

59.3 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 60

Schema binding

The Rust binding of the language-independent schema model, and the check it performs at open.

60.1 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 `Durable`, 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 `Durable` type declares its own descriptor in `describe_local`, and the derive macro generates that declaration.

60.2 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.

60.3 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 structural descriptors, so that a tool can decode them: a `Vec<T>` and a `DurableVec<T>` alike a `Sequence` of the element type, a `String` `Text`, and a map a `Map` of its key type and its value type. An enum's variant holds one payload: `()`, the empty struct, for a unit variant, the field's type for a variant of one unnamed field, and otherwise an anonymous struct of its fields named after the variant (the derive). `Option<T>` declares an enum `Option` with a one-byte discriminant width, `None` = 0 holding `()` and `Some` = 1 holding a `T`; a tuple a struct `TupleN` with positional fields; `Fixed<T>` a `Fixed` of `T`'s descriptor; and `Box<T>` is *schema-transparent*, reusing `T`'s descriptor, as `#[kladde(transparent)]`

types do. Only `DurableBlob` declares `Opaque`, without a fixed size, since its serde payload is not decomposable.

60.4 At create and at open

`Kladde::create` encodes the root type's descriptor table into the schema chain, which the header's `schema_head` 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 — decoding by the writer's descriptors — does not exist yet, so a mismatch cannot be bridged; see Evolution.

60.5 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 61

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.

61.1 The interface

`kladde-store` is a persistent store of one byte sequence, usable on its own: `Store` implements `WriteBackend`, whose methods are the journal's records — `write`, `splice`, `move_range` and `paste` — and the transaction calls, plus `len`. `write` and `splice` return the cut of what they removed, an `Option<CutId>`, and `splice` records a `Write` wherever the lengths agree. A cut lives as long as the journal that made it: `is_alive` says whether that journal is still unfolded, and `paste` takes the cut's bytes along with it, which it records as a `Paste` while the cut lives and as an ordinary splice of those bytes after. `Store::flush`, `flush_if_journal_exceeds`, `read_all`, `schema` and `set_schema`, `private` and `set_private`, and `stats` complete it; the schema and private chains are byte strings the next flush writes.

61.2 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 `Mut`s 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 whose transaction would not fit in what is left of the journal's 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.

61.3 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.

61.4 The mirror

Table pages stay resident as the reference algorithms recommend, each a boxed page-sized array keyed by page number. Data pages are read into a mirror while a file is opened, in page order, and dropped once the sequence is decoded; a flush that needs a span's bytes again, to move, merge or rewrite it, reads them from the file through a small cache.

61.5 The in-memory structures

61.5.1 The span table

A `Vec<Span>` indexed by span id, of **48 bytes per span** rather than the 16 the reference algorithms count, since each record also holds what the store keeps beside the span table in the language-agnostic design: the span's chain, its rope node, its heat, its entry's size and its place in its table page, and its place in its data page's reverse list. A bitmap of used ids beside it serves the id allocator, which hands out the id after the predecessor's, if it is free, and the lowest free id otherwise — which keeps the spans a flush makes one after another consecutive without looking for a run.

61.5.2 The treap

The rope is an implicit treap, not a counted B-tree: a randomized binary tree in an arena of nodes, each knowing its parent and the total length below it, `O(log k)` expected steps per operation and 40 bytes per node. It is there for ease of implementation: a split at a position and a join of two trees are all that the rope, the fold's piece table and the journal's cuts need, so one arena type of under 400 lines serves all three, while splitting and joining whole B-trees at a position is the intricate part of a counted B-tree. Its priorities come from a seeded generator, so runs reproduce. With the treap's node, a span costs about 90 bytes in memory, against the 32 of the reference algorithms; for the files measured that is a fraction of a megabyte, and a B-tree is the better choice once memory per span matters.

61.5.3 Maps keyed by id or page

Sets and maps keyed by span id or page number use a multiplicative hasher rather than `std`'s SipHash, whose protection against HashDoS a map built from the file does not need. The page table is a `Vec` of an enum over the page states, indexed by page number, since lowest-first placement keeps page numbers dense; only a live page has an epoch and a victim bucket.

61.5.4 The journal buffer

Every record since the last flush lives in one `VecDeque<Record>` beside a byte arena, also a `VecDeque`, that holds their payloads, with positions counted from the start of the session (one buffer and three cursors). A record holds absolute positions in memory and is delta-encoded only when it is appended.

61.6 Where the integer widths come from

Every width below is forced by the specification's bounds, not chosen:

quantity	width	because
span id	`u32`	span ids are 32-bit, and 0 means none
span length	`u32`	a `Zero` span may be up to `2^32 - 1` bytes
page offset	`u16`	page sizes are at most 64 KiB

quantity	width	because
page number	`u32`	page numbers are 32-bit
position in the sequence	`u64`	the sequence is shorter than `2^48` bytes
a size in a container's index	`u32`	four bytes per element; a sum of 4 GiB or more panics today, where it should fail cleanly (implementation notes)

Chapter 62

kladde-cpp

The C++ implementation of kladde: one of the two reference implementations of the specification, independent of kladde-rs in Rust, and the library behind the Inkscape demo.

Everything here is **specific to C++**. The file format and the algorithms are shared with kladde-rs; the description of types, the handles and the error handling are not. Its design was worked out in the C++ draft, which this section records as built, with the places where building it went another way.

62.1 What it is

A CMake project, C++20, with no required dependencies: a compiled schema layer and store, and a header-only typed layer on top.

```
#include <kladde/kladde.hpp>

struct Contact {
    std::string email;
    kladde::Vector<PhoneNumber> phones;
    bool starred = false;
};
KLADDE_STRUCT(Contact, email, phones, starred)

auto file = kladde::File<Contact>::create("contacts.kladde", Contact{.email = "ada@example.com"}).valu
Contact const& contact = file.get();   // reading is plain C++
auto c = file.edit();                  // kladde::Mut<Contact>
c.starred().set(true);                 // one Write of one byte
c.email().insert(0, "lady.");          // one Splice
c.phones().push_back(Mobile{"555-0100"});  // one transaction
```

62.2 Three layers

layer	headers	holds
the schema	`kladde/schema.hpp`	descriptors, their canonical encoding, fingerprints, sizes, and a textual dump for comparing tables across implementations
the store	`kladde/store.hpp`	the file format, the journal, the flush, consolidation, and storage; positions and bytes only
typed values	`kladde/kladde.hpp`	which C++ types are durable, how they encode, their handles, and `kladde::File`

The store and the schema are compiled into `kladde::kladde`; the typed layer is templates over a small virtual interface, `kladde::Backend`, which the store implements and which a `kladde::VectorBackend` — a plain byte vector that applies every mutation at once — implements for tests.

62.3 Which C++ types are durable

Standard types are durable wherever the in-memory value needs nothing a standard type cannot hold, and kladde's own containers take over where it does.

C++	kind	handle
`std::uint8_t` … `std::uint64_t`, `std::int8_t` … `std::int64_t`, `float`, `double`, `bool`, `char32_t`	Primitive	`set`
`std::string`	Text	`set` (records only the changed middle), `insert`, `erase`, `replace`, `append`, `clear`
`std::pair`, `std::tuple`	Struct `TupleN`	`get<I>()`, `parts()`
`std::array<T, N>`	Array	`at(i)`
`std::optional<T>`	Enum `Option`	`emplace`, `reset`, `value()`
`std::vector<T>` for fixed-size `T`	Sequence	`at`, `insert`, `push_back`, `erase`, `pop_back`, `clear`
`std::variant<…>`	Enum, named with `KLADDE_VARIANT` or `KLADDE_VARIANT_NAMED`	`set`, `visit`, `if_holds<A>`
`enum class` with `KLADDE_ENUM`	Enum of unit variants	`set`
structs with `KLADDE_STRUCT`	Struct	an accessor per field, `at<&T::m>()`, `parts()`
structs with `KLADDE_TRANSPARENT`	as their one member	the member's
`kladde::Vector<T>`	Sequence	`at`, `insert`, `push_back`, `remove`, `move`, `cut`, `insert_cut`
`kladde::List<T>`	Sequence	as `Vector`, plus `detach` and `insert_element`; long-lived `kladde::Ref<T>`
`kladde::Map<K, V>`	Map	`at(key)`, `insert`, `remove`
`kladde::Box<T>`	as `T`	`T`'s
`kladde::Fixed<T>` for scalars and `enum class`	Fixed	`set`, one Write

Text must be UTF-8. A `std::string` used as a byte buffer — file contents, a serialized message, a binary identifier — is not text, and storing one fails; hold bytes in a `std::vector<std::uint8_t>`, a Sequence of `u8`.

A `std::variant`'s variants are named after its alternatives' types, and its discriminants are their indices, unless `KLADDE_VARIANT(V, "Name", 1, 2, …)` pins the discriminants or `KLADDE_VARIANT_NAMED(V, "Name", (0, "Circle"), (1, "Poly"), …)` also names the variants. Variant names are part of a type's fingerprint, so a program that shares files with one written in another language names them as that program does: Rust's `Poly(Vec<Point>)` is held in C++ as a `std::vector<Point>`, which would otherwise be named after its type.

A `std::vector` of elements whose size depends on their value fails to compile, with a message that names `kladde::Vector`: it is the one place where the plain type would quietly cost what kladde exists to avoid.

62.4 Handles

`kladde::Mut<T>` is the short-lived handle, kladde-rs's `Mut`, valid until something is written through a handle that is not itself or made from it. C++ cannot enforce that rule, so it is documented, like iterator invalidation; and since a handle made from another refers to its parent's place in the sequence, **a handle must not be moved while a handle made from it lives**.

`kladde::Ref<T>` is the long-lived handle, to an element of a `kladde::List`, valid until the element is destroyed, as Values and their positions describes. Each list element is an *anchor*: it knows its list and its index, and finds its start by asking its list, which asks the anchor of the value that holds it, up to the file's root. The list finds its own offset inside that value by walking the value's fields to it, which costs the sum of the sizes of the fields before it, each of which every durable type knows in constant time.

62.5 Errors

Opening and creating return `kladde::expected<T>`, a C++20 stand-in for `std::expected<T, kladde::error>`, since their failures are ordinary and the caller must handle them, with or without exceptions.

A mutation or a flush fails only when the disk does, and the file is then poisoned. With exceptions, the call throws `kladde::exception`; without, it calls the handler that `kladde::set_error_handler` installs and returns, leaving the value as it was, since memory changes only after the store recorded the change.

62.6 Flushing

Flushes happen by themselves as the journal fills, and on `flush()` and `close()`. An interactive application calls `file.flush_if_journal_exceeds(0.5)` from its idle callback, so that a flush, and its `fsync`, falls between gestures; the Inkscape demo does so from a low-priority `Glib::signal_idle` handler after each change. As in kladde-rs, a write or splice of at least `Options::payload_threshold` bytes keeps them in payload pages, a flush seals the journal it folds where a reusable page can start the next one, and `Options::sync_on_worker` lets a flush that a mutation triggered run its `fsync` and header write on a `std::thread` of their own, as Transactions and batches describes; it is off by default.

62.7 Inside the library

Inside the library, every fallible function returns an `expected`, and `KLADDE_TRY` propagates it as Rust's `?` does, so the port of the store follows kladde-rs line by line.

62.8 Where kladde-cpp departs from the draft

- **Struct fixed forms are not implemented yet**: `kladde::Fixed<T>` takes scalars and `enum class` only.
- **`kladde::Box<T>` reports no fixed size**, where kladde-rs's `Box<T>` reports `T`'s: a recursive type reaches itself through a box, and asking would make the compiler instantiate the type inside itself. The fixed size is an optimization of the typed layer, not part of the schema, so files are unaffected; only a `std::vector` of boxes is refused that kladde-rs would accept.
- **An `enum class`'s discriminant width in the fixed form is the smallest of 1, 2, 4 and 8 bytes that holds the largest discriminant**, as for a Rust enum without `#[repr]`, not the size of the underlying type, which C++ defaults to `int`.
- **Class hierarchies (`kladde::Poly<Base>`), debug checking of handles, and reflection are not built yet.**
- **The Inkscape demo observes Inkscape's XML tree rather than replacing it**; see the evaluation.

Chapter 63

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".

63.1 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 `id → address` table, and sizedness as a placement hint	Pages and the address table, since superseded too	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
Allocations and the address table — allocations with ids and owning pointers, statements keyed by `(id, offset)` resolved per byte, and slotted and packed places	The sequence and the span table	the description was paid per allocation and a splice per allocation's tail, so neither could be kept small without the other growing; one sequence cut into spans keyed by id unties the two
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, since superseded too	keying by byte offset instead of by allocation made per-range statements viable, which removed the need for every mechanism in the left column

superseded design	replaced by	what changed
Sizes from extents — an allocation's size inferred from the extents of its statements, resizes as a bound or a fence, and zeros by default	The newest size wins, since superseded too	nothing in the constraints: a writer always knew the size, so the inference bought nothing, and its machinery hid failure modes that a realistic workload exposed

63.2 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 64

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.

64.1 What it was

The snapshot was a **heap**: a partition of `[0, 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 `id → 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.

64.2 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>);
}
```

64.2.1 The heap owned the id table

The naive division of labour would put free-space management in an "allocator" and the `id → 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.

64.2.2 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.

64.2.3 A step was a byte range, not an allocation

```rust
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.

64.2.4 `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.

64.3 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 `UniquePointer-FixedSize`, 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.

64.4 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.

64.5 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.

64.6 `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 65

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**.

65.1 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.

65.2 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 Φ $\iff$ 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.

65.3 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.

65.4 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.

65.4.1 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.

65.5 Pathologies

65.5.1 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.

65.5.2 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.

65.5.3 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.

65.5.4 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:

```
count × L ≥ T · (s + r_len)     and     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.

65.6 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.

65.7 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 66

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.

66.1 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.

66.2 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.

66.3 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.

66.4 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 67

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.

67.1 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.

67.2 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.

67.3 The mechanisms it needed, and no longer does

67.3.1 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`.

67.3.2 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.

67.3.3 A separate index

The **index** was a small rooted structure that made scanning unnecessary: a list of `(page_number, epoch)` for every live `Table` 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 ≈ 8 MB ≈ 2000 table pages; at ≈ 6 bytes per index entry that is ≈ 12 KB ≈ 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.

67.3.4 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.

67.4 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 68

Sizes from extents

Superseded. Replaced by the newest size wins and the coverage rule.

An earlier address table let a reader infer an allocation's size from its statements instead of reading it from one: every content statement's extent counted toward the size, and a resize was stated by one of two statements, one that only bounded the size from below and one that also fenced off everything older beyond its bound. Bytes that no statement matched read as zero by default.

68.1 Why it was abandoned

The reader's inference bought a flexibility that no writer ever needed, and the machinery that kept it correct hid failure modes until tests ran into them. A writer always knows the size it wants, yet the size depended on what was physically present, including statements in leaves that no in-memory structure reaches. Keeping it right took a per-allocation record of the fencing statement, which every page rewrite had to transfer; a second kind of pin for statements that only bounded the size; a fencing statement at every shrink, since whether one was needed could not be decided; and a rule against both resize statements in one epoch, which led kladde-rs to refuse nearly empty leaves as consolidation fillers, so that they piled up on a realistic workload. Six gaps in this machinery surfaced only once kladde-rs implemented and tested it.

Its replacement has the writer state the size whenever it changes, mostly at the cost of one bit on a content statement that ends there, and state the zeros a growth exposes, which kladde's own data types wrote anyway.

Chapter 69

Allocations and the address table

Superseded. Replaced by the sequence and the span table.

The design between the relocatable heap and the present one: the file's content as allocations that pointed at each other, described on copy-on-write pages by an address table of statements keyed by an allocation's id and an offset in it. Implemented in full by kladde-rs, whose branch `address-table` holds it as it stood, measured in most of the evaluation pages, and specified in detail; kladde-docs' history holds every document of it, up to commit `bf13c73`. What follows is its outline, and why it was replaced.

69.1 What it was

Allocations. An allocation was an untyped byte range with a stable 32-bit id, a size, and content; ids were never zero, which left zero for a null pointer. A value lived in an *inline slot* of a fixed `INLINE_SIZE` inside some allocation, and a value with content of variable length — a string, a vector, a map — owned allocations of its own, reached through a pointer serialized as the target's id. Every allocation had exactly one owning pointer, so the value graph was a tree of ownership; dropping or overwriting an owning value freed what it owned, recursively, in the same transaction and after the write that superseded the pointer.

The address table was a list of *statements* in table pages, each inheriting its page's epoch: `Ref(id, offset, size, address)`, `Zero(id, offset, size)`, and `Inline(id, offset, size, payload)` stated content, each with a *sizing* variant that also said the allocation ended where the content ended; `Size(id, n)` stated a size alone, and `Tombstone(id)` ended an allocation's existence. Statements of different epochs were resolved per byte, the newest winning wherever it applied, and an allocation's size was its newest size statement's. A writer had to keep the *coverage rule* — every byte that became part of an allocation was stated in that epoch — or truncated content could reappear.

The journal recorded effects on allocations: `Free`, `Resize`, `Write`, `Splice`, `Copy` and `Move`, each but `Free` bringing its allocation into existence. Every mutation decomposed as *prepare new, unreachable state → one publishing write → clean up what it replaced*, an ordering discipline no type system enforced.

In memory, a loaded file was a *fragment map* from `(id, offset)` to the resolved content, a *statement slab* counting each live statement's *pins* — the fragments, sizes and tombstones that kept it alive — an *allocation map* of sizes, fragment counts and `mentions`, and the page table. A flush folded the journal into a piece table per allocation, took the fragments it changed, and derived every statement from one dirty set at the cut; consolidation added a fourth mechanism, *description defragmentation*, which rewrote stippled ranges of an allocation as one statement, materializing the zeros between them.

Values came in *slotted* and *packed* places, small values kept short content inline, and the Rust binding had `Location`, `UniquePointer`, `store`, `load` and `free`, with one hole through which `load` could mint a second owner.

69.2 Why it was replaced

The description was paid per allocation and a splice per allocation's tail, so keeping either small made the other grow. A small description wanted values inline in few allocations, and then each allocation was long, and a splice into one restated or rewrote everything behind it, since statements were keyed by their offset in the allocation: on kladde-svg's small layout of the world map, flushes were measured to restate 24 bytes of statements per edit that had lain in leaves and to write 16 bytes of data again, growing with the length of the lists. Cheap splices wanted short allocations, and then each value paid its pointer and statements, a cost that grew with the number of values rather than with their bytes. No policy escaped that, because the format tied the unit of description to the unit of the data model. The one-sequence design untied them: the description depends only on how a writer cuts one sequence into spans, and spans are keyed by id, so a splice restates the spans at its cuts and nothing behind them. The file as one sequence, measured compares the two on the same drawings and texts.

69.3 Compared with the present design

	allocations and the address table	one sequence
description written per splice	every statement behind it in its allocation, or its bytes written again	one to three entries
moving a value within a container	the moved bytes, or `Move`'s statement edits	three to six spans, no bytes
moving a value between containers	its pointers written again, its allocations left where they are	a cut and a paste in one transaction: three to six spans, no bytes
description written per overwrite	one statement	three entries the first time, one after
description per value	a pointer, its statements, and its size	none; a few bytes per span of up to a page
what a page rewrite must also state	the sizes and tombstones of the ids it touches	nothing
liveness	pins over fragments, size pins, tombstones, `mentions`	the newest entry of a reached span
the store's memory	fragment map, statement slab, allocation map, page table	span table, rope, page table, reverse index
removing a value	a splice, then a free walk over what it owns	a splice
field access when mutating	constant offsets	constant up to the first variable-size field
opening	a sweep resolving bytes across epochs	one pass over entries, a chain walk, and the decode
upgrading to a new schema	the allocations holding values whose layout changed, written again	the values whose encoding changed, spliced in again
writing in the writer's schema	offsets from the writer's inline sizes	offsets and encodings interpreted from the writer's descriptors
parts of a file in different schema versions	not supported; a capsule would take an allocation	not supported; a capsule would be inline
tooling	opaque containers, a pointer graph	a decode from the schema

What the present design pays for it: containers keep positions in memory, four bytes per variable-size

element; offsets after a value whose size depends on its value are computed at run time; making a guard costs a lookup in its parent's index and a change of size an update per level; an overwrite inside a long span splits it; every insertion into a sequence takes two records, the element and the count; moving a value between containers holds a transaction open; and partial loading needs segments, which need a pointer of their own.

69.4 The vocabulary that went with it

term	why it went	instead
allocation, allocation id, allocation size	there are none	the sequence, and a value's region
pointer, owning pointer, ownership, `UniquePointer`, `Location`	nothing points at anything	a region; a segment's reference, later
free, freeing, free walk, leak	nothing is allocated	removing, deleting: a splice
tombstone, size statement, sizing variant, `Size`	no size is stated	—
coverage rule	no byte becomes part of an allocation	the reach rule
fragment, fragment map, stipple, shadowing of part of a statement	no entry shows part of a span	span, run of spans; an entry replaces another whole
pin, `mentions`, statement slab, allocation map, recyclable set	liveness is one bit	live entry, reached span
`Resize`, `Free`, `Copy`	nothing to act on	—
slotted and packed places; inline size, `INLINE_SIZE`	a type has one encoding	fixed-size, variable-size, `Fixed(T)`, encoded size
small value, spilling	everything is inline	—
root allocation, schema table, consolidator state, as allocations	they are chains	the sequence, the schema chain, the private chain
address table, statement	entries describe spans whole	span table, entry
description defragmentation	there is no stipple to fill	merging spans