FREE SAMPLE

Introduction & Unit 1 of 8

Build a SQL Database Engine in C++

Through Challenges

Parts I & II

Storage, and From Bytes to a Query

Hatem M.

Contents

What this book is **1**

The thesis, stated plainly **1**

The path: fourteen units, four parts **1**

What this book does *not* cover **2**

Who should read this **3**

The reference machine **3**

Building the code **3**

How each challenge is structured **4**

Conventions **4**

The contract **4**

Part I **5**

Unit 1 — Slotted Pages and the Pager **6**

Where we are. . . . 6
What this unit builds 6
The engine so far 6
Challenge 1.1 — A page you can trust 6
Challenge 1.2 — The slotted page: records inside a page 12
Challenge 1.3 — Erase, fragmentation, and compaction 15
Challenge 1.4 — The pager: pages that outlive the process 22
Challenge 1.5 — Allocation and the free list 30
Challenge 1.6 — The experiment: why pages, measured 36
Where we landed 41
The engine so far 42
Run it yourself 43

What comes next **44**

What this book is

This book builds a working SQL database engine in C++20, from an empty directory to a query processor that runs real SQL against data on disk, and it does so through a single relentless method: **nothing is asserted; everything is demonstrated.**

Every data structure is built, compiled, and run. Every performance claim is a table printed by a benchmark whose source is on the page. Every design decision is followed by the measurement that justifies it — and, where the design has a cost, by the measurement that exposes that cost too. When a component fails, it fails in front of you: each unit contains one **forensic trap**, a real bug reproduced deterministically, diagnosed from the evidence, fixed, and locked shut with a regression test. The book is called *Through Challenges* because you are meant to try each problem before you read its solution. The solutions are complete and compilable; the understanding is yours to earn.

This is a lab manual, not a survey. You will not find hand-waving about how B+Trees are "generally logarithmic" — you will find the fan-out arithmetic, the page-read counts, and a benchmark that walks a tree of a million keys. The tone throughout is that of a rigorous engineering notebook: precise, honest about limitations, and allergic to unverified claims.

The thesis, stated plainly

A database engine is not magic and not a monolith. It is a stack of comprehensible layers — pages on disk, a tree to index them, a pool to cache them, a language to query them, a planner to choose how — and each layer can be built and *measured* by one person who is willing to be rigorous. By the end you will not just know how a database works; you will have proof, in compilable code and reproducible numbers, of why each piece is built the way it is.

The path: fourteen units, four parts

The engine — we call it **labdb** — grows monotonically. Each unit adds to the same codebase; nothing is thrown away except in the deliberate, signposted moments when a later challenge replaces an earlier file with a more capable version.

Unit	Part I — Storage	Delivers
1	Slotted Pages & the Pager	self-describing 4 KiB pages, records with stable ids, a pager with a free list
2	B+Tree: Insert & Search	logarithmic lookup — the answer to Unit 1's scan problem
3	B+Tree: Delete & Range Scans	deletion, rebalancing, ordered range queries
4	The Buffer Pool	caching hot pages in memory, with eviction

Unit	Part II — From Bytes to a Query	Delivers
5	The Record Layer	typed rows: a schema-aware codec and a table heap, reached by key through the index

Unit	Part II — From Bytes to a Query	Delivers
6	The Catalog	persistent schemas and table names — the engine's self-knowledge
7	The Front End: Tokenizer & Parser	SQL text into an abstract syntax tree, with caret-pointed errors
8	The Executor	**milestone:** end-to-end `SELECT`/`INSERT` on real data

Unit	Part III — Making It Fast & Real	Delivers
9	Joins	nested-loop vs hash join, with the 100× experiment
10	Aggregation	`GROUP BY`, `COUNT`, `SUM`, and friends
11	Indexes & the Planner	using B+Trees to plan queries — and when an index *loses*
12	Vectorized Execution	revisiting Unit 8's executor, measured in nanoseconds per row

Unit	Part IV — Durability & Capstone	Delivers
13	Write-Ahead Log & Crash Recovery	surviving the torn write from Unit 1; a kill-9 recovery harness
14	Capstone	correctness verified against PostgreSQL, performance benched against SQLite

Each unit opens by taking honest inventory — what exists, what is missing, what hurts — and closes with what was measured, what is now known, and the one specific pain that motivates the next unit. The through-line is a chain of measured problems, each solved by the next layer.

What this book does *not* cover

An intro that promised everything would deliver nothing. To keep every claim demonstrable, the following are explicitly out of scope — deferred to a possible second volume, and flagged wherever the main text brushes against them:

- **Concurrency and MVCC.** labdb is single-threaded. Multi-version concurrency control, locking, and isolation levels are a book of their own.
- **A wire protocol.** No client/server layer; the engine is exercised in-process and through tests.
- **The full SQL grammar.** We implement a rigorous, useful subset — enough to run real queries end to end — not the entire standard.
- **External-memory sorting** and an **advanced cost-based optimizer.** The planner in Unit 11 is real but deliberately simple; industrial query optimization is out of scope.

These exclusions are a feature. Every page of this book is something you can compile, run, and measure — and that promise is only keepable with a bounded scope.

Who should read this

You should be comfortable reading and writing C++ — not necessarily an expert, but not meeting `std::vector` for the first time. You do not need prior database internals knowledge; the book builds that from the pages up. Some familiarity with the shell and a Unix-like system will help, since the measurements assume POSIX.

If you want to go deeper on adjacent skills, three companion books pair naturally with this one. The **SQL Mastery Series** covers SQL from the query-writer's side — the language this engine executes. **C++ Autopsy** dissects the language mechanics (undefined behavior, aliasing, the cost of abstractions) that this book leans on but does not pause to fully derive; where the main text says "readers of *C++ Autopsy* will recognize this," it is pointing to a fuller treatment there. And **Build an LLM Inference Engine in C++** applies the same build-it-and-measure-it method to a very different system, for readers who like this approach and want to see it turned on another hard problem.

The reference machine

Every number in this book comes from running the shown code on one machine. Being specific about it is part of being honest about the measurements:

- **Compiler:** g++ 13.3.0, `-std=c++20 -O2 -Wall -Wextra`
- **OS / kernel:** Ubuntu 24.04 LTS, Linux 6.x
- **CPU:** one virtualized x86-64 core (Intel Xeon class, ~2.1 GHz)
- **Memory:** ~4 GiB
- **Storage:** ext4 on a virtio block device

That this is a **virtualized** host matters, and the book says so at every measurement it affects. Absolute I/O timings — especially `fsync` latency and anything labeled "cold cache" — reflect the virtualization layer and would differ, sometimes greatly, on bare metal with a physical disk. For that reason the book consistently leans on **ratios and counts** (syscalls per scan, the fsync-to-write multiple, warm-cache speedups) rather than absolute microseconds, because those ratios are properties of the software and reproduce across machines. Where an absolute number is contaminated by the environment, you will see it flagged, not buried. Every table is labeled with the command that produced it; run that command on your machine and you will get your machine's version of the same story.

Building the code

The complete source accompanies the book, organized so that every listing you read is something the build system actually compiles:

```
src/        the engine, growing one unit at a time
tests/      per-unit correctness tests and measurement programs
tools/      forensic-trap demos and inspection utilities
snapshots/  mid-unit versions of files that later challenges replace,
            kept so that every listing in the book compiles as printed
scripts/    one-command reproduction of each unit's results
```

From the repository root, three commands do everything:

```
$ make all      # build the engine, tests, benchmarks, and demos
$ make test     # run every correctness test
$ make bench    # run every measurement
```

There are no third-party dependencies — only a C++20 compiler and a POSIX system. Everything compiles cleanly under `-Wall -Wextra`; a warning is treated as a defect, not a suggestion.

How each challenge is structured

Every challenge in the book follows the same shape, and it is designed to be *attempted*, not just read:

1. **The goal** — what we are building and why it matters now.
2. **The challenge** — the precise specification, with the design tension made explicit. *Stop here and try it.*
3. **The hint** — the key idea, for when you want a nudge rather than the answer.
4. **The solution** — complete, compilable C++20. Not a sketch.
5. **Why it works** — the reasoning, including complexity analysis and the costs the design accepts.
6. **The measurement** — the command, the code, the table of real numbers, and an honest interpretation.

The book rewards a reader who treats step 2 as an assignment. You will learn more from ten minutes of trying and failing to store variable-length records in a fixed-size page than from any amount of reading about slot directories.

Conventions

A few conventions hold throughout, so the code reads consistently:

- **Naming.** Types are `PascalCase`, functions and variables `snake_case`, compile-time constants `kCamelCase`, and data members carry a `trailing_underscore_`.
- **Error policy.** *Expected* outcomes — a full page, an absent key — travel through return values (`std::optional`, `bool`) and never throw. *Broken invariants and failed syscalls* are unrecoverable: they throw and the process stops. A storage engine that limps past a failed write is more dangerous than one that halts.
- **On-disk format.** Every byte is explicit and little-endian, read and written with `memcpy` — no struct overlays, no alignment assumptions. The format is a contract, and the code treats it like one, refusing to build on a big-endian host rather than claim an untested portability.
- **Reproducibility.** Every benchmark uses a fixed random seed. Your timings will vary; your counts, capacities, and the exact bytes of every reproduced bug will not.

The contract

Here is the promise this book makes to you: **every line of code compiles, every test passes, and every performance number is real — produced by running the code you can see on the machine described above.** Nothing is asserted; everything is demonstrated. Hold the book to it.

Turn the page. We start with an empty directory and the lowest question there is: the operating system hands you a file full of bytes, and SQL wants to ask for a row. Everything begins there.

Part I

Storage — pages, an index, and a cache

Unit 1 — Slotted Pages and the Pager

Where we are

What exists. An empty directory and a compiler. That is the honest starting inventory, and it is worth staring at for a second, because every database you have ever used began as exactly this.

What is missing. Everything — but "everything" is not a plan. What is missing *first* is an answer to the lowest-level question a storage engine faces: the operating system hands you a file, which is a growable array of bytes with no structure, no types, no records, and no opinions. SQL, when we get to it in Part II, will ask questions like "give me row 481,116 of `users`." Between those two worlds — a byte array and a row — sits every decision this unit makes.

What hurts. Nothing yet, because nothing exists. By the end of this unit something will exist, and it will hurt in one very specific, measurable way. That pain is Unit 2's reason to exist.

What this unit builds

The universal answer to "how do engines structure a file" — used by PostgreSQL, SQLite, InnoDB, and now by us — is the **page**: the file is an array of fixed-size blocks, each block is self-describing, and *all* I/O happens in whole blocks. This unit builds that answer from bytes up:

- **Challenge 1.1** — a `Page`: 4,096 raw bytes with a byte-exact, self-describing header. *(creates `src/common.hpp`, `src/page.hpp`)*
- **Challenge 1.2** — the slotted layout: variable-length records inside a page, addressable by stable slot ids. *(creates `src/slotted_page.hpp`)*
- **Challenge 1.3** — erase, fragmentation, and compaction: making a page survive churn. *(replaces `src/slotted_page.hpp`)*
- **Challenge 1.4** — the `Pager`: moving pages between memory and one database file with `pread`/`pwrite`, plus this unit's **forensic trap**: the torn write. *(creates `src/io.hpp`, `src/io.cpp`, `src/pager.hpp`, `src/pager.cpp`)*
- **Challenge 1.5** — page allocation and the free list. *(replaces `src/pager.hpp`, `src/pager.cpp`)*
- **Challenge 1.6** — the experiment pages exist for: per-record file I/O versus paged I/O, measured cold and warm.

One scoping note so nobody waits for something that is not coming: in this unit a *record* is an opaque span of bytes. The engine does not know about columns, types, or NULLs yet — typed rows appear starting in Unit 5, whose record layer gives bytes a schema to be decoded against; Unit 6's catalog then persists that schema and gives the table a name. Storage first, meaning later.

The engine so far

file	lines	role
—	0	nothing. That is the point.

Every unit from here on opens with this manifest grown a little larger.

Challenge 1.1 — A page you can trust

The goal

Define the engine's atom: a `Page` of exactly 4,096 bytes carrying a 20-byte header that describes what the page is, and typed accessors for that header that are byte-exact, portable within our stated constraints, and free of undefined behavior. Also fix, once and for the whole book, the engine's error-handling policy.

Why 4,096? It is the page size of essentially every filesystem and MMU we will run on, it divides evenly into the 512 B/4 KiB sectors disks actually read and write, and it is what SQLite defaults to and PostgreSQL doubles (8 KiB). We *choose* it; nothing in this unit's code would change at 8 KiB or 16 KiB, and by the time we can measure B+Tree fan-out (Unit 2) you will be equipped to question the choice with data.

The challenge

Create `src/common.hpp` and `src/page.hpp` providing:

1. `kPageSize = 4096`, a `PageId` type (`u32`), and `kNullPage = 0` — page 0 will be reserved for engine metadata in Challenge 1.4, so id 0 can double as "no page" in on-disk links.
2. A `Page` class holding exactly `kPageSize` bytes — `static_assert` it — that default-constructs to all zeroes.
3. This header, at these offsets, little-endian, no exceptions:

offset	size	field	meaning
0	4	`page_id`	the page's own id (self-check against file position)
4	2	`page_type`	invalid = 0, meta = 1, slotted = 2, free = 3
6	2	`slot_count`	slots in the directory (Challenge 1.2)
8	2	`free_start`	first byte past the slot directory
10	2	`free_end`	first byte of the cell area
12	2	`frag_bytes`	dead bytes inside the cell area (Challenge 1.3)
14	2	reserved	zero; a later unit will claim it
16	4	`next_page`	generic link: free list now, leaf chains in Unit 2

4. Get/set accessors for every field that compile to plain loads and stores, **without** casting a struct over the buffer.
5. An error policy: expected outcomes travel through return values; broken invariants and failed syscalls throw and kill the process.

Constraint worth respecting before you peek: field access must not rely on struct layout, must not require aligned pointers, and must not violate strict aliasing. There is exactly one standard-blessed tool for this.

Hint

`std::memcpy` into and out of a local integer is defined behavior for any byte source, aligned or not, and every optimizing compiler since the last decade turns a 2- or 4-byte fixed-size `memcpy` into a single move instruction. Write `load_u16`/`store_u16`/`load_u32`/`store_u32` helpers once in `common.hpp`; make each accessor a one-liner over a `constexpr` offset. For endianness: decide, assert, move on — C++20 gives you `std::endian::native` to `static_assert` against.

The solution

Two files are created. First `src/common.hpp` — constants, the error policy, and the byte-access helpers everything else in this book will use:

```
#pragma once
// labdb -- common definitions shared by every layer of the engine.
```

```cpp
// Introduced in Challenge 1.1.

#include <bit>
#include <cerrno>
#include <cstdint>
#include <cstring>
#include <stdexcept>
#include <string>

namespace labdb {

// The engine's one fixed size. Everything on disk is a multiple of this.
inline constexpr std::size_t kPageSize = 4096;

// Pages are named by 32-bit ids. Page 0 is the pager's meta page and is
// never handed out, so 0 doubles as "no page" in on-disk links.
using PageId = std::uint32_t;
inline constexpr PageId kNullPage = 0;

// The on-disk format is little-endian, byte for byte. We refuse to build on
// a big-endian host rather than pretend to a portability we have not tested.
static_assert(std::endian::native == std::endian::little,
              "labdb's on-disk format assumes a little-endian host");

// ---------------------------------------------------------------------------
// Error policy (fixed here, used for the whole book):
//   * Expected outcomes -- page full, slot empty, key absent -- travel
//     through return values (std::optional, bool). Never exceptions.
//   * Broken invariants and failed syscalls are unrecoverable: we throw,
//     the process reports and dies. A storage engine that limps past a
//     failed write is worse than one that stops.
// ---------------------------------------------------------------------------

[[noreturn]] inline void fail(const std::string& msg) {
  throw std::runtime_error(msg);
}

// For syscall results: attaches strerror(errno) to the message.
inline void check_sys(bool ok, const char* what) {
  if (!ok) fail(std::string(what) + ": " + std::strerror(errno));
}

// For internal invariants.
inline void check_that(bool ok, const char* what) {
  if (!ok) fail(std::string("invariant violated: ") + what);
}

// ---------------------------------------------------------------------------
// Byte-exact field access. All on-disk integers are read and written with
// memcpy: no struct overlays, no alignment assumptions, no strict-aliasing
// violations. Challenge 1.1 measures what this safety costs (nothing).
// ---------------------------------------------------------------------------

inline std::uint16_t load_u16(const std::uint8_t* p) {
  std::uint16_t v;
  std::memcpy(&v, p, sizeof v);
  return v;
}

inline std::uint32_t load_u32(const std::uint8_t* p) {
  std::uint32_t v;
  std::memcpy(&v, p, sizeof v);
  return v;
}

inline void store_u16(std::uint8_t* p, std::uint16_t v) {
  std::memcpy(p, &v, sizeof v);
}

inline void store_u32(std::uint8_t* p, std::uint32_t v) {
  std::memcpy(p, &v, sizeof v);
}

}  // namespace labdb
```

And the page itself, `src/page.hpp` — bytes plus a typed view of the header, and deliberately nothing else. Interpretations of the *payload* (records today, B+Tree nodes in Unit 2) will be separate view classes; keeping `Page` dumb is what lets one buffer pool serve every page type later.

```cpp
#pragma once
// A Page is kPageSize raw bytes plus typed access to the 20-byte header.
// It knows nothing about records. Interpretations of the payload -- slotted
// records in this unit, B+Tree nodes in Unit 2 -- are separate view classes.
// Introduced in Challenge 1.1.

#include <array>
#include <cstdint>

#include "common.hpp"

namespace labdb {

enum class PageType : std::uint16_t {
  kInvalid = 0,  // all-zero page: allocated but never formatted
  kMeta    = 1,  // page 0 only: pager bookkeeping (Challenge 1.4)
  kSlotted = 2,  // variable-length records (Challenge 1.2)
  kFree    = 3,  // on the free list (Challenge 1.5)
};

// Header layout, byte-exact:
//
//   offset  size  field       meaning
//   ------  ----  ----------  ----------------------------------------
//        0     4  page_id     who am I (self-check against file offset)
//        4     2  page_type   PageType
//        6     2  slot_count  slots in the directory, live or dead
//        8     2  free_start  first byte past the slot directory
//       10     2  free_end    first byte of the cell area
//       12     2  frag_bytes  dead bytes inside the cell area
//       14     2  reserved    zero for now; a later unit will claim it
//       16     4  next_page   generic link: free list now, leaf chains later
//
inline constexpr std::size_t kPageHeaderSize = 20;

class Page {
 public:
  Page() : bytes_{} {}  // all zeroes: PageType::kInvalid

  std::uint8_t*       data()       { return bytes_.data(); }
  const std::uint8_t* data() const { return bytes_.data(); }

  PageId id() const       { return load_u32(data() + 0); }
  void set_id(PageId v)   { store_u32(data() + 0, v); }

  PageType type() const   { return static_cast<PageType>(load_u16(data() + 4)); }
  void set_type(PageType v) { store_u16(data() + 4, static_cast<std::uint16_t>(v)); }

  std::uint16_t slot_count() const     { return load_u16(data() + 6); }
  void set_slot_count(std::uint16_t v) { store_u16(data() + 6, v); }

  std::uint16_t free_start() const     { return load_u16(data() + 8); }
  void set_free_start(std::uint16_t v) { store_u16(data() + 8, v); }

  std::uint16_t free_end() const       { return load_u16(data() + 10); }
  void set_free_end(std::uint16_t v)   { store_u16(data() + 10, v); }

  std::uint16_t frag_bytes() const     { return load_u16(data() + 12); }
  void set_frag_bytes(std::uint16_t v) { store_u16(data() + 12, v); }

  PageId next_page() const     { return load_u32(data() + 16); }
  void set_next_page(PageId v) { store_u32(data() + 16, v); }

 private:
  alignas(64) std::array<std::uint8_t, kPageSize> bytes_;
};

static_assert(sizeof(Page) == kPageSize,
              "a Page must be exactly one page of bytes -- nothing more");

}  // namespace labdb
```

Why it works

Why not `reinterpret_cast<Header*>(bytes)`? Three separate reasons, any one of which is disqualifying for an on-disk format. *Layout:* the standard does not promise a struct has no padding between members; our offsets must be contractual, not incidental. *Aliasing:* accessing bytes through an unrelated

struct type is undefined behavior, and optimizers genuinely exploit that assumption. *Alignment:* a `u32` member would require 4-byte alignment the buffer position cannot always guarantee once records start at arbitrary offsets. `memcpy` has none of these problems: its behavior is defined for any bytes, any alignment — and, as the measurement below shows, it costs literally zero instructions. (Readers of *C++ Autopsy* will recognize this trio; here we only need the verdict.)

Why explicit little-endian? Because an on-disk format is a contract with the future, and "whatever the CPU felt like" is not a contract. We assert little-endian at compile time and refuse to build elsewhere — an honest limitation stated in code, rather than a portability claim we never tested.

Why a self-id in the header? `read_page(17)` returning a page that *claims* to be page 12 is the symptom of an offset-arithmetic bug, and with the self-check (Challenge 1.4) it is caught at the moment of the read, not three units later as a corrupted B+Tree. Twenty bytes of header per 4,096 is 0.5% overhead; a page that can identify itself is the cheapest insurance this book buys.

Complexity. Every accessor is O(1): one fixed-size copy the compiler lowers to one instruction. There is nothing asymptotic to discuss yet — which is itself the point of starting here.

The measurement

Two claims were made: the layout is byte-exact, and the safety is free. Both get demonstrated, not asserted.

Claim 1 — byte-exactness. `tests/u01_page_layout_test.cpp` (109 lines, in the repository) writes every field and then inspects the raw buffer at the documented offsets, least-significant byte first, including the edge value `free_end = 4096` and the untouched reserved bytes:

```
$ ./bin/u01_page_layout_test

u01_page_layout_test: PASS (all header fields at documented offsets)
```

Claim 2 — the safety is free. The benchmark sums all 2,048 `u16` values in a page 200,000 times — 409.6 million loads — once through `load_u16` and once through the raw pointer cast we refused to use (present in the benchmark only, clearly labeled as UB-for-comparison):

```cpp
// Challenge 1.1 measurement: what does memcpy-based field access cost
// compared to a raw pointer cast? Sum all 2048 u16 values in a page,
// 200,000 times, both ways.
#include <cstdint>
#include <cstdio>

#include "common.hpp"
#include "harness.hpp"

using namespace labdb;
using labdb::test::do_not_optimize;
using labdb::test::now_s;

namespace {

alignas(64) std::uint8_t g_buf[kPageSize];

// The engine's way: load_u16 (memcpy inside).
std::uint64_t sum_via_memcpy() {
  std::uint64_t sum = 0;
  for (std::size_t i = 0; i < kPageSize; i += 2) sum += load_u16(g_buf + i);
  return sum;
}

// The "fast, unsafe" way. This violates strict aliasing and exists in this
// benchmark only so we can measure what the shortcut buys. It buys nothing.
std::uint64_t sum_via_cast() {
  std::uint64_t sum = 0;
  const auto* w = reinterpret_cast<const std::uint16_t*>(g_buf);
  for (std::size_t i = 0; i < kPageSize / 2; ++i) sum += w[i];
  return sum;
}

}  // namespace
```

```
int main() {
  for (std::size_t i = 0; i < kPageSize; ++i)
    g_buf[i] = static_cast<std::uint8_t>(i * 37 + 11);

  // Correctness first: both must agree.
  if (sum_via_memcpy() != sum_via_cast()) {
    std::fprintf(stderr, "FAIL: sums disagree\n");
    return 1;
  }

  constexpr int kRounds = 200000;
  const double total = double(kRounds) * double(kPageSize / 2);

  std::uint64_t acc = 0;
  double t0 = now_s();
  for (int r = 0; r < kRounds; ++r) {
    acc += sum_via_memcpy();
    do_not_optimize(acc);
  }
  const double memcpy_s = now_s() - t0;

  t0 = now_s();
  for (int r = 0; r < kRounds; ++r) {
    acc += sum_via_cast();
    do_not_optimize(acc);
  }
  const double cast_s = now_s() - t0;
  do_not_optimize(acc);

  std::printf("| method | u16 loads | seconds | ns per load |\n");
  std::printf("|---|---:|---:|---:|\n");
  std::printf("| `load_u16` (memcpy) | %.0f | %.3f | %.4f |\n", total, memcpy_s,
              memcpy_s / total * 1e9);
  std::printf("| pointer cast (UB) | %.0f | %.3f | %.4f |\n", total, cast_s,
              cast_s / total * 1e9);
  return 0;
}
```

On the reference machine:

```
$ ./bin/u01_header_bench

| method | u16 loads | seconds | ns per load |
|---|---:|---:|---:|
| `load_u16` (memcpy) | 409600000 | 0.121 | 0.2951 |
| pointer cast (UB) | 409600000 | 0.094 | 0.2298 |
```

Honesty requires reading that table carefully: the two *loops* are not identical — 0.23 versus 0.30 nanoseconds per load. At these speeds both loops are vectorized and retiring multiple loads per cycle; the gap is the compiler choosing slightly different unrolling for the two loop shapes, which is a fact about *summing 2,048 values in a row*, not about a single field access. The engine never sums headers; it loads one field at a time. So the evidence that actually matters is the generated code for exactly that operation, isolated in `tools/u01_header_asm.cpp`:

```
// Two ways to load a u16 from a byte buffer, isolated so the generated
// code can be inspected:
//   g++ -std=c++20 -O2 -S -o - tools/u01_header_asm.cpp
// (The cast version violates strict aliasing; it exists for comparison.)
#include <cstdint>
#include <cstring>

std::uint16_t load_via_memcpy(const std::uint8_t* p) {
  std::uint16_t v;
  std::memcpy(&v, p, sizeof v);
  return v;
}

std::uint16_t load_via_cast(const std::uint8_t* p) {
  return *reinterpret_cast<const std::uint16_t*>(p);
}
```

```
$ g++ -std=c++20 -O2 -S -o - tools/u01_header_asm.cpp | grep -A6 'load_via'
```

```
_Z15load_via_memcpyPKh:
.LFB31:
    .cfi_startproc
    endbr64
    movzwl    (%rdi), %eax
    ret
    .cfi_endproc
--
_Z13load_via_castPKh:
.LFB32:
    .cfi_startproc
    endbr64
    movzwl    (%rdi), %eax
    ret
    .cfi_endproc
```

One `movzwl`, one `ret` — instruction-for-instruction identical. The defined-behavior version and the undefined-behavior version compile to the same machine code; the only difference is that one of them is allowed to keep working when you change compilers. This is the cheapest lesson in the book: **measure before you compromise on correctness, because the payment you were bracing for is often zero.** It is also a preview of a discipline we will need constantly: a microbenchmark measures the loop you wrote, not the abstraction you care about — corroborate it at the instruction level when the claim matters.

Files after this challenge: `src/common.hpp` (76 lines), `src/page.hpp` (71 lines).

Challenge 1.2 — The slotted page: records inside a page

The goal

Store variable-length records inside a single page and hand each one a **stable slot id** that survives the movement of other records. Reading slot *s* must return exactly the bytes written to slot *s*, and the id must not change when neighbors come and go. This is the interface every layer above will address storage through, so its guarantees matter more than its cleverness.

The challenge

Create `src/slotted_page.hpp` defining a `SlottedPage` that *views* a `Page` (it holds a `Page*`, stores no bytes of its own) and provides:

- `init(id)` — format an empty slotted page.
- `insert(bytes) -> optional<slot>` — append a record; return its slot id, or `nullopt` if the page is full. Full is an expected outcome, not an error: it returns, it does not throw.
- `read(slot) -> optional<span>` — return a view of the record's bytes.

The design problem is the layout. Records are variable-length, so you cannot index them by multiplication the way you would a fixed-size array. And you must not move a record's *identity* when you reclaim space, or every pointer the rest of the engine holds would dangle.

Hint

This is the **slotted page**, and its trick is indirection through a directory that grows from one end of the page while records ("cells") grow from the other:

```
[ header 20B ][ slot dir --> ]      free      [ <-- cells ]
^0           ^20         ^free_start   free_end^          ^4096
```

A slot is 4 bytes — a `u16` offset and a `u16` length. The slot id is an *index into the directory*, so it is stable no matter where the cell physically sits: to move a record you rewrite its 4-byte slot, never its id. The page is full when the directory and the cell area would collide, i.e. when `free_end - free_start` cannot fit the record plus one new slot. Growing from both ends means one contiguous free region in the middle, so "does it fit" is a single subtraction.

The solution

This is the Challenge 1.2 version of `src/slotted_page.hpp` — insert and read only. Challenge 1.3 will add erase and compaction and replace the file; everything here survives that change unaltered.

```cpp
#pragma once
// SlottedPage -- interprets a Page's payload as a set of variable-length
// records addressed by stable slot ids. Challenge 1.2: insert and read.
//
// Layout inside the page:
//
//   [ header 20B ][ slot directory --> ]  free  [ <-- cell area ]
//   ^0            ^kPageHeaderSize   ^free_start ^free_end       ^4096
//
// The directory grows forward, cells grow backward; the page is full when
// they would meet. A slot is 4 bytes: u16 cell offset, u16 cell length.

#include <cstdint>
#include <cstring>
#include <optional>
#include <span>

#include "page.hpp"

namespace labdb {

inline constexpr std::size_t kSlotSize = 4;
inline constexpr std::size_t kMaxRecordSize =
    kPageSize - kPageHeaderSize - kSlotSize;  // 4072: one record, one slot

class SlottedPage {
 public:
  explicit SlottedPage(Page& page) : p_(&page) {}

  // Format the underlying page as a fresh, empty slotted page.
  void init(PageId id) {
    p_->set_id(id);
    p_->set_type(PageType::kSlotted);
    p_->set_slot_count(0);
    p_->set_free_start(static_cast<std::uint16_t>(kPageHeaderSize));
    p_->set_free_end(static_cast<std::uint16_t>(kPageSize));
    p_->set_frag_bytes(0);
    p_->set_next_page(kNullPage);
  }

  std::uint16_t slot_count() const { return p_->slot_count(); }

  // Bytes between the end of the directory and the start of the cell area.
  std::size_t contiguous_free() const {
    return static_cast<std::size_t>(p_->free_end()) - p_->free_start();
  }

  // Insert a record. Returns its slot id, or nullopt if it does not fit.
  std::optional<std::uint16_t> insert(std::span<const std::uint8_t> rec) {
    if (rec.empty() || rec.size() > kMaxRecordSize) return std::nullopt;
    const auto len = static_cast<std::uint16_t>(rec.size());
    if (contiguous_free() < len + kSlotSize) return std::nullopt;  // full

    const auto off = static_cast<std::uint16_t>(p_->free_end() - len);
    std::memcpy(p_->data() + off, rec.data(), len);
    p_->set_free_end(off);

    const std::uint16_t slot = slot_count();
    p_->set_slot_count(static_cast<std::uint16_t>(slot + 1));
    p_->set_free_start(
        static_cast<std::uint16_t>(p_->free_start() + kSlotSize));
    set_slot(slot, off, len);
    return slot;
  }

  // Read a record by slot id. The span points into the page: it is valid
  // only until the next mutation of this page.
  std::optional<std::span<const std::uint8_t>> read(std::uint16_t slot) const {
    if (slot >= slot_count()) return std::nullopt;
    return std::span<const std::uint8_t>(p_->data() + slot_off(slot),
                                         slot_len(slot));
  }

 private:
  static constexpr std::size_t slot_pos(std::uint16_t slot) {
    return kPageHeaderSize + kSlotSize * static_cast<std::size_t>(slot);
  }
  std::uint16_t slot_off(std::uint16_t s) const {
    return load_u16(p_->data() + slot_pos(s));
```

```
  }
  std::uint16_t slot_len(std::uint16_t s) const {
    return load_u16(p_->data() + slot_pos(s) + 2);
  }
  void set_slot(std::uint16_t s, std::uint16_t off, std::uint16_t len) {
    store_u16(p_->data() + slot_pos(s), off);
    store_u16(p_->data() + slot_pos(s) + 2, len);
  }

  Page* p_;
};

}  // namespace labdb
```

Why it works

Stable ids for free. The slot id is an index into the directory, and `insert` only ever *appends* a slot, so a live record's id is fixed for the lifetime of the page. When Challenge 1.3 moves cells around to reclaim space, it rewrites offsets inside existing slots — the ids the rest of the engine holds never move. This is the whole reason to pay four bytes of indirection per record: it is what lets Unit 2 point a B+Tree leaf at a slot and trust the pointer.

One subtraction to check fit. Because the directory grows up and cells grow down, the free space is always the single contiguous run `[free_start, free_end)`. No free-list walk, no first-fit scan: `insert` needs `len + 4` bytes and either has them or does not. That simplicity is worth protecting, and it is exactly what fragmentation will threaten in the next challenge.

The view/page split. `SlottedPage` stores only a `Page*`. It is a lens, not an owner — which means the same 4,096 bytes can be interpreted as a slotted page here and, in Unit 2, as a B+Tree node by a different view class, and in Unit 4 a buffer pool can own the bytes while views come and go. Keeping interpretation separate from storage is a decision that pays off for the rest of the book.

Complexity. `insert` and `read` are both O(1): a fit check, a memcpy of the payload, a slot write. No loop runs over existing records. (That changes precisely once, for dead-slot reuse in Challenge 1.3, and we will measure the consequence rather than hand-wave it.)

The bytes-are-opaque boundary, restated. `insert` takes a span and `read` returns one. `SlottedPage` never looks *inside* a record. That is not laziness; it is the layering that lets the same page code carry untyped bytes now and typed, schema-decoded rows from Unit 5 on, with no change here.

The measurement

The claim to demonstrate is capacity: how many records of a given size fit in one 4,096-byte page, and how much of the page is payload versus overhead. `tests/u01_slotted_test.cpp` (the full listing appears in Challenge 1.3, since it also exercises erase) fills a page with fixed-size records until `insert` returns `nullopt`, verifies every record reads back byte-identical, and reports the counts:

```
$ ./bin/u01_slotted_test
records per page by record size (page 4096 B, header 20 B, slot 4 B):

| record size | records/page | payload bytes | payload utilization |
|---:|---:|---:|---:|
| 16 B | 203 | 3248 | 79.3% |
| 64 B | 59 | 3776 | 92.2% |
| 256 B | 15 | 3840 | 93.8% |
| 1024 B | 3 | 3072 | 75.0% |
```

Read the pattern, because it explains a design tension we will meet again. Each record costs its own bytes plus 4 for its slot, against a fixed budget of `4096 - 20 = 4076` bytes. For 16-byte records the 4-byte slot is a 25% tax on each entry, so utilization sags to 79%. At 64 and 256 bytes the slot is noise and we keep 92–94%. At 1,024 bytes something different bites: three records need $3\times(1024+4)=3{,}084$ bytes and a fourth would need 4,112 — over budget — so the page holds three and 24% simply cannot be filled. Large records waste space to *quantization*, not to per-record overhead. Hold on to this: it is the first appearance of the fan-out arithmetic that decides B+Tree height in Unit 2, where the same "how many entries fit in a page" question sets how many disk reads a lookup costs.

Challenge 1.3 — Erase, fragmentation, and compaction

The goal

Make a page survive *churn*. Add `erase(slot)` so records can be deleted, reclaim the space they leave behind, and keep every invariant intact through an unbounded sequence of interleaved inserts and erases — without ever invalidating the slot id of a record that is still live.

The challenge

Replace `src/slotted_page.hpp` with a version that adds:

- `erase(slot) -> bool` — delete a record, returning `false` if the slot was already dead.
- Automatic space reclamation, so that after erasing records a new `insert` can reuse the freed bytes even when they are not contiguous.
- Dead-slot reuse, so a workload that erases and inserts in equal measure does not grow the directory forever.

The subtlety: when you erase a record from the *middle* of the cell area, you leave a hole. Holes are not immediately reusable, because cells are packed against the end of the page and allocation only knows how to carve from the one contiguous free region. You need a way to turn scattered holes back into contiguous space, a rule for when to do it, and — the part that is easy to get wrong — an invariant that stays true the entire time.

Hint

Track dead space instead of chasing it. Add a `frag_bytes` counter (already reserved at offset 12) for bytes trapped in holes. `erase` marks the slot dead (offset 0 — which lands inside the header, so no live cell can have it) and adds the cell's length to `frag_bytes`. Then split the "does it fit" decision in two: if the record fits in the *contiguous* free region, business as usual; if it fits only in contiguous-plus-fragmented, run `compact()` first — walk the directory, copy live cells tight against the end of the page through a scratch buffer, reset `frag_bytes` to zero — then insert into the now-contiguous space. If it does not fit even then, the page is genuinely full. Two more touches keep it honest: reuse a dead slot before growing the directory, and let dead slots at the very tail of the directory give their bytes back. The invariant that must never break:

```
(bytes from free_end to end of page) == (live record bytes) + frag_bytes
```

The solution

The full, final `src/slotted_page.hpp` — this is the version the rest of the book uses:

```
#pragma once
// SlottedPage -- interprets a Page's payload as a set of variable-length
// records addressed by stable slot ids. Challenges 1.2 (insert/read) and
// 1.3 (erase/compaction).
//
// Layout inside the page:
//
//   [ header 20B ][ slot directory --> ]  free  [ <-- cell area ]
//   ^0            ^kPageHeaderSize   ^free_start ^free_end       ^4096
//
// The directory grows forward, cells grow backward; the page is full when
// they would meet. A slot is 4 bytes: u16 cell offset, u16 cell length.
// offset == 0 marks a dead slot -- offset 0 lands inside the header, so no
// live cell can ever be there.

#include <array>
#include <cstdint>
#include <cstring>
#include <optional>
#include <span>

#include "page.hpp"

namespace labdb {

inline constexpr std::size_t kSlotSize = 4;
inline constexpr std::size_t kMaxRecordSize =
```

```
    kPageSize - kPageHeaderSize - kSlotSize;  // 4072: one record, one slot

class SlottedPage {
 public:
  explicit SlottedPage(Page& page) : p_(&page) {}

  // Format the underlying page as a fresh, empty slotted page.
  void init(PageId id) {
    p_->set_id(id);
    p_->set_type(PageType::kSlotted);
    p_->set_slot_count(0);
    p_->set_free_start(static_cast<std::uint16_t>(kPageHeaderSize));
    p_->set_free_end(static_cast<std::uint16_t>(kPageSize));
    p_->set_frag_bytes(0);
    p_->set_next_page(kNullPage);
  }

  std::uint16_t slot_count() const { return p_->slot_count(); }

  // Bytes between the end of the directory and the start of the cell area.
  std::size_t contiguous_free() const {
    return static_cast<std::size_t>(p_->free_end()) - p_->free_start();
  }

  // Contiguous space plus dead bytes that compaction can reclaim.
  std::size_t total_free() const {
    return contiguous_free() + p_->frag_bytes();
  }

  bool live(std::uint16_t slot) const {
    return slot < slot_count() && slot_off(slot) != 0;
  }

  // Insert a record. Returns its slot id, or nullopt if the record cannot
  // fit even after compaction. Compacts automatically when the space
  // exists but is fragmented (Challenge 1.3).
  std::optional<std::uint16_t> insert(std::span<const std::uint8_t> rec) {
    if (rec.empty() || rec.size() > kMaxRecordSize) return std::nullopt;
    const auto len = static_cast<std::uint16_t>(rec.size());

    // Reuse a dead slot if one exists: it costs no directory bytes and
    // keeps the directory from growing without bound under churn.
    std::optional<std::uint16_t> reuse;
    const std::uint16_t n = slot_count();
    for (std::uint16_t i = 0; i < n; ++i) {
      if (slot_off(i) == 0) {
        reuse = i;
        break;
      }
    }

    const std::size_t need = len + (reuse ? 0 : kSlotSize);
    if (contiguous_free() < need) {
      if (total_free() < need) return std::nullopt;  // genuinely full
      compact();  // space exists, just fragmented: rebuild the cell area
    }

    const auto off = static_cast<std::uint16_t>(p_->free_end() - len);
    std::memcpy(p_->data() + off, rec.data(), len);
    p_->set_free_end(off);

    std::uint16_t slot;
    if (reuse) {
      slot = *reuse;
    } else {
      slot = n;
      p_->set_slot_count(static_cast<std::uint16_t>(n + 1));
      p_->set_free_start(
          static_cast<std::uint16_t>(p_->free_start() + kSlotSize));
    }
    set_slot(slot, off, len);
    return slot;
  }

  // Read a record by slot id. The span points into the page: it is valid
  // only until the next mutation of this page. (Unit 4's forensic trap is
  // what happens to people who forget this.)
  std::optional<std::span<const std::uint8_t>> read(std::uint16_t slot) const {
    if (!live(slot)) return std::nullopt;
    return std::span<const std::uint8_t>(p_->data() + slot_off(slot),
```

```
                                        slot_len(slot));
  }

  // Erase a record. Its bytes become fragmentation; its slot becomes dead
  // and reusable. Live slot ids are never invalidated by an erase.
  bool erase(std::uint16_t slot) {
    if (!live(slot)) return false;
    p_->set_frag_bytes(
        static_cast<std::uint16_t>(p_->frag_bytes() + slot_len(slot)));
    set_slot(slot, 0, 0);
    // Dead slots at the tail of the directory can give their bytes back.
    std::uint16_t n = slot_count();
    while (n > 0 && slot_off(static_cast<std::uint16_t>(n - 1)) == 0) {
      --n;
      p_->set_free_start(
          static_cast<std::uint16_t>(p_->free_start() - kSlotSize));
    }
    p_->set_slot_count(n);
    return true;
  }

  // Rebuild the cell area so it contains live cells only, packed against
  // the end of the page. O(kPageSize) time, one page of scratch space.
  void compact() {
    std::array<std::uint8_t, kPageSize> scratch;
    auto write_pos = static_cast<std::uint16_t>(kPageSize);
    const std::uint16_t n = slot_count();
    for (std::uint16_t i = 0; i < n; ++i) {
      const std::uint16_t off = slot_off(i);
      if (off == 0) continue;  // dead slot: its cell is not copied
      const std::uint16_t len = slot_len(i);
      write_pos = static_cast<std::uint16_t>(write_pos - len);
      std::memcpy(scratch.data() + write_pos, p_->data() + off, len);
      set_slot(i, write_pos, len);
    }
    std::memcpy(p_->data() + write_pos, scratch.data() + write_pos,
                kPageSize - write_pos);
    p_->set_free_end(write_pos);
    p_->set_frag_bytes(0);
  }

 private:
  static constexpr std::size_t slot_pos(std::uint16_t slot) {
    return kPageHeaderSize + kSlotSize * static_cast<std::size_t>(slot);
  }
  std::uint16_t slot_off(std::uint16_t s) const {
    return load_u16(p_->data() + slot_pos(s));
  }
  std::uint16_t slot_len(std::uint16_t s) const {
    return load_u16(p_->data() + slot_pos(s) + 2);
  }
  void set_slot(std::uint16_t s, std::uint16_t off, std::uint16_t len) {
    store_u16(p_->data() + slot_pos(s), off);
    store_u16(p_->data() + slot_pos(s) + 2, len);
  }

  Page* p_;
};

}  // namespace labdb
```

Why it works

The invariant is the whole design. Every mutation is written to preserve one equation: the cell area (`kPageSize - free_end`) always equals live payload bytes plus `frag_bytes`. `insert` into contiguous space grows the cell area and the live total together, leaving `frag_bytes` untouched. `erase` moves a cell's bytes from the live total into `frag_bytes` — the cell area is unchanged, the record is simply reclassified from live to dead. `compact()` copies only live cells and zeroes `frag_bytes`, shrinking the cell area by exactly the dead bytes it dropped. Because the equation holds after each operation, the "fits only after compaction" test (`total_free = contiguous_free + frag_bytes`) is always exactly right — the measurement below verifies this across 300,000 random operations.

Compaction is amortized, not constant, and we schedule it lazily. A `compact()` is O(page size) — it can touch every live byte. If we compacted on every erase, a delete-heavy workload would pay that

cost constantly. Instead we compact only when an insert genuinely needs the space, so the cost is amortized across the inserts that made the page fragmented. The measurement quantifies exactly how often it fires and what each one costs.

Dead-slot reuse bounds the directory. Without it, a workload that repeatedly erases one record and inserts another would append a fresh slot every insert while dead slots piled up — the directory would march toward the cell area and choke the page. Scanning for a dead slot before growing is the one O(slot count) loop in `insert`; on a 4 KiB page the directory is at most a few hundred entries, and trimming dead slots off the tail keeps it short in the common case. We pay a bounded scan to bound the directory — a trade the churn benchmark shows is decisively worth it.

Why the span from `read` is borrowed, not owned. `read` returns a `span` into the page's own bytes; the next mutation of that page may move or overwrite those bytes. Callers must copy if they need the data to outlive the next `insert`/`erase`/`compact`. This is a deliberate zero-copy/lifetime trade — and it is a loaded gun that Unit 4's forensic trap fires, when a cached page moves under a pointer someone kept too long. Noted here; paid for there.

The measurement

Two things need demonstrating: that the invariants actually hold under sustained random churn, and that compaction costs what the theory says.

Invariants under churn. `tests/u01_slotted_test.cpp` runs 300,000 randomized operations — 55% inserts, 45% erases — against a `std::map` reference model, re-checking the full invariant set (directory geometry, the cell-area equation, and byte-exact readback of every live record) every 5,000 operations. Here is the complete test, which also produced the capacity table in Challenge 1.2:

```
// Challenges 1.2 + 1.3 correctness.
// Phase 1: deterministic fill/read for four record sizes (also produces
//          the records-per-page table).
// Phase 2: 300,000 randomized insert/erase operations checked against an
//          in-memory reference model, with page invariants verified
//          throughout. Seed is fixed: every run is the same run.
#include <cstdint>
#include <cstdio>
#include <cstring>
#include <map>
#include <random>
#include <span>
#include <vector>

#include "harness.hpp"
#include "page.hpp"
#include "slotted_page.hpp"

using namespace labdb;

namespace {

using Model = std::map<std::uint16_t, std::vector<std::uint8_t>>;

std::vector<std::uint8_t> make_record(std::uint32_t seed, std::size_t len) {
  std::vector<std::uint8_t> r(len);
  for (std::size_t i = 0; i < len; ++i)
    r[i] = static_cast<std::uint8_t>(seed * 2654435761u + i * 97);
  return r;
}

// The page invariants that must hold after every operation.
void check_invariants(Page& page, const Model& model) {
  SlottedPage view(page);
  REQUIRE(page.free_start() ==
          kPageHeaderSize + kSlotSize * page.slot_count());
  REQUIRE(page.free_start() <= page.free_end());
  REQUIRE(page.free_end() <= kPageSize);

  std::size_t live_bytes = 0;
  for (const auto& [slot, rec] : model) live_bytes += rec.size();
  // The cell area is exactly live bytes plus dead (fragmented) bytes.
  REQUIRE(kPageSize - page.free_end() == live_bytes + page.frag_bytes());

  // Every record the model knows about reads back byte-identical.
```

```
  for (const auto& [slot, rec] : model) {
    auto got = view.read(slot);
    REQUIRE(got.has_value());
    REQUIRE(got->size() == rec.size());
    REQUIRE(std::memcmp(got->data(), rec.data(), rec.size()) == 0);
  }
}

}  // namespace

int main() {
  // ---- Phase 1: fill with fixed-size records, read everything back ----
  std::printf(
      "records per page by record size (page %zu B, header %zu B, slot %zu B):\n\n",
      kPageSize, kPageHeaderSize, kSlotSize);
  std::printf("| record size | records/page | payload bytes | payload utilization |\n");
  std::printf("|---:|---:|---:|---:|\n");
  for (std::size_t size : {std::size_t{16}, std::size_t{64}, std::size_t{256},
                           std::size_t{1024}}) {
    Page page;
    SlottedPage view(page);
    view.init(1);
    std::uint32_t count = 0;
    while (true) {
      auto rec = make_record(count, size);
      if (!view.insert(rec).has_value()) break;
      ++count;
    }
    for (std::uint16_t s = 0; s < count; ++s) {
      auto got = view.read(s);
      auto expect = make_record(s, size);
      REQUIRE(got && got->size() == size);
      REQUIRE(std::memcmp(got->data(), expect.data(), size) == 0);
    }
    std::printf("| %zu B | %u | %zu | %.1f%% |\n", size, count,
                std::size_t{count} * size,
                100.0 * double(count) * double(size) / double(kPageSize));
  }

  // ---- Phase 2: randomized churn against a reference model ----
  Page page;
  SlottedPage view(page);
  view.init(7);
  Model model;
  std::mt19937 rng(42);
  std::uniform_int_distribution<std::size_t> len_dist(8, 256);

  constexpr int kOps = 300000;
  int inserts = 0, erases = 0, full_rejections = 0, compactions = 0;
  std::uint32_t next_seed = 1000;

  for (int op = 0; op < kOps; ++op) {
    const bool do_insert = model.empty() || (rng() % 100) < 55;
    if (do_insert) {
      auto rec = make_record(next_seed++, len_dist(rng));
      const std::uint16_t frag_before = page.frag_bytes();
      auto slot = view.insert(rec);
      if (slot) {
        // frag > 0 collapsing to 0 across an insert means compact() ran.
        if (frag_before > 0 && page.frag_bytes() == 0) ++compactions;
        REQUIRE(model.count(*slot) == 0);  // never hands out a live slot
        model[*slot] = std::move(rec);
        ++inserts;
      } else {
        // "Full" must be honest: even compaction could not have made room.
        const bool has_dead_slot = view.slot_count() > model.size();
        const std::size_t need = rec.size() + (has_dead_slot ? 0 : kSlotSize);
        REQUIRE(view.total_free() < need);
        ++full_rejections;
      }
    } else {
      auto it = model.begin();
      std::advance(it, rng() % model.size());
      REQUIRE(view.erase(it->first));
      REQUIRE(!view.read(it->first).has_value());  // really gone
      model.erase(it);
      ++erases;
    }
    if (op % 5000 == 0) check_invariants(page, model);
  }
```

```
  check_invariants(page, model);

  std::printf("\nrandomized model test: PASS\n");
  std::printf(
      "  ops=%d inserts=%d erases=%d page-full rejections=%d compactions=%d\n",
      kOps, inserts, erases, full_rejections, compactions);
  std::printf("  live records at end=%zu, frag_bytes=%u, contiguous_free=%zu\n",
              model.size(), page.frag_bytes(), view.contiguous_free());
  return 0;
}
```

Running it exercises both phases — the capacity table from 1.2 and the randomized churn — in one shot:

```
$ ./bin/u01_slotted_test

records per page by record size (page 4096 B, header 20 B, slot 4 B):

| record size | records/page | payload bytes | payload utilization |
|---:|---:|---:|---:|
| 16 B | 203 | 3248 | 79.3% |
| 64 B | 59 | 3776 | 92.2% |
| 256 B | 15 | 3840 | 93.8% |
| 1024 B | 3 | 3072 | 75.0% |

randomized model test: PASS
  ops=300000 inserts=135118 erases=135091 page-full rejections=29791 compactions=37569
  live records at end=27, frag_bytes=253, contiguous_free=29
```

The first block is the capacity table from Challenge 1.2; the tail is the churn result that matters here. 135,118 inserts and 135,091 erases churned through a single 4,096-byte page with 37,569 compactions and 29,791 honest page-full rejections, and the `std::map` shadow agreed on every read-back at every checkpoint. The page survives arbitrary churn with its invariants intact. That is the correctness claim discharged; now the cost.

What compaction costs. `tests/u01_churn_bench.cpp` measures two things: sustained erase+insert throughput on one page, and the cost of a single `compact()` as a function of how many live bytes it must move.

```
// Challenge 1.3 measurement.
// Part 1: sustained erase+insert churn on one page -- throughput, and how
//         often compaction actually fires.
// Part 2: the cost of one compact() as a function of live bytes.
#include <cstdint>
#include <cstdio>
#include <cstring>
#include <random>
#include <span>
#include <vector>

#include "harness.hpp"
#include "page.hpp"
#include "slotted_page.hpp"

using namespace labdb;
using labdb::test::do_not_optimize;
using labdb::test::now_s;

int main() {
  // ---- Part 1: churn ----
  {
    Page page;
    SlottedPage view(page);
    view.init(1);
    std::mt19937 rng(1234);
    std::uniform_int_distribution<std::size_t> len_dist(16, 192);
    std::vector<std::uint8_t> buf(256, 0xAB);

    std::vector<std::uint16_t> live;
    while (true) {  // fill to capacity with random-size records
      auto slot =
          view.insert(std::span<const std::uint8_t>(buf.data(), len_dist(rng)));
      if (!slot) break;
      live.push_back(*slot);
    }
```

```
  constexpr int kOps = 2000000;  // one op = one erase + one insert
  long compactions = 0;
  const double t0 = now_s();
  for (int i = 0; i < kOps; ++i) {
    const std::size_t victim = rng() % live.size();
    view.erase(live[victim]);
    live[victim] = live.back();
    live.pop_back();

    std::size_t len = len_dist(rng);
    for (;;) {
      const std::uint16_t frag_before = page.frag_bytes();
      auto slot =
          view.insert(std::span<const std::uint8_t>(buf.data(), len));
      if (slot) {
        // frag > 0 collapsing to 0 across an insert means compact() ran.
        if (frag_before > 0 && page.frag_bytes() == 0) ++compactions;
        live.push_back(*slot);
        break;
      }
      len /= 2;            // page too full for this size: try smaller
      if (len < 8) break;  // give up on this insert
    }
  }
  const double dt = now_s() - t0;
  do_not_optimize(live);

  std::printf(
      "churn on one page: %d erase+insert pairs in %.3f s -> %.0f ns per pair\n",
      kOps, dt, dt / kOps * 1e9);
  std::printf("compactions triggered: %ld (one every %.1f pairs)\n\n",
              compactions,
              double(kOps) / double(compactions ? compactions : 1));
}

// ---- Part 2: compact() cost vs live bytes ----
std::printf("| target fill | live bytes | live records | ns per compact() |\n");
std::printf("|---:|---:|---:|---:|\n");
for (int fill_pct : {25, 50, 75, 95}) {
  Page page;
  SlottedPage view(page);
  view.init(1);
  std::vector<std::uint8_t> rec(32, 0xCD);
  const std::size_t target = kPageSize * std::size_t(fill_pct) / 100;
  std::vector<std::uint16_t> slots;
  std::size_t live_bytes = 0;
  while (live_bytes + rec.size() <= target) {
    auto s = view.insert(rec);
    if (!s) break;
    slots.push_back(*s);
    live_bytes += rec.size();
  }
  // Fragment it: erase every other record. What is left is what
  // compact() must move.
  std::size_t erased = 0;
  for (std::size_t i = 0; i < slots.size(); i += 2) {
    view.erase(slots[i]);
    ++erased;
  }
  live_bytes -= erased * rec.size();
  const std::size_t live_records = slots.size() - erased;

  Page snapshot;
  std::memcpy(snapshot.data(), page.data(), kPageSize);

  constexpr int kIters = 50000;
  double t0 = now_s();
  for (int i = 0; i < kIters; ++i) {
    std::memcpy(page.data(), snapshot.data(), kPageSize);  // restore frag
    view.compact();
    do_not_optimize(page.data()[kPageSize - 1]);
  }
  const double with_compact = now_s() - t0;

  t0 = now_s();
  for (int i = 0; i < kIters; ++i) {
    std::memcpy(page.data(), snapshot.data(), kPageSize);
    do_not_optimize(page.data()[kPageSize - 1]);
  }
```

```
    const double restore_only = now_s() - t0;

    const double ns = (with_compact - restore_only) / kIters * 1e9;
    std::printf("| %d%% | %zu | %zu | %.0f |\n", fill_pct, live_bytes,
                live_records, ns);
  }
  return 0;
}
```

```
$ ./bin/u01_churn_bench

churn on one page: 2000000 erase+insert pairs in 0.429 s -> 215 ns per pair
compactions triggered: 845010 (one every 2.4 pairs)

| target fill | live bytes | live records | ns per compact() |
|---:|---:|---:|---:|
| 25% | 512 | 16 | 91 |
| 50% | 1024 | 32 | 162 |
| 75% | 1536 | 48 | 249 |
| 95% | 1792 | 56 | 281 |
```

Two results worth reading closely. First, sustained churn runs at about 215 nanoseconds per erase+insert pair even though a compaction fires roughly every 2.4 pairs — because most of those compactions are moving a nearly-empty page's handful of live bytes. That is amortization working exactly as designed: the expensive operation is frequent only when it is cheap. Second, the cost of a single `compact()` climbs with live bytes — from 91 ns at 512 live bytes to 281 ns at 1,792 — and the climb is essentially linear at about 0.14 ns per live byte. That is the O(live bytes) prediction confirmed by measurement, not asserted from the code: compaction copies live cells and nothing else, so its cost tracks live data, not page size or hole count. We now have a page that reclaims its own space, at a price we have measured rather than guessed.

Files after this challenge: `src/slotted_page.hpp` grows to 167 lines; `src/common.hpp` and `src/page.hpp` unchanged.

Challenge 1.4 — The pager: pages that outlive the process

The goal

Give pages a home on disk. Build a `Pager` that opens one database file, reads and writes whole pages by id at their natural byte offsets, and guarantees that a page written before the process exits is a page the next process can read back. Along the way, establish the file's identity (a magic number and a reserved meta page) and — most important for this book — confront what "written to disk" actually promises.

The challenge

Create `src/io.hpp`, `src/io.cpp`, `src/pager.hpp`, and `src/pager.cpp`. This challenge builds the open/read/write/sync core; Challenge 1.5 adds allocation and the free list. Requirements:

1. **Page 0 is the meta page.** It holds a magic string (so we can reject files that are not ours), the page count, and — starting in 1.5 — the free-list head. It never leaves the pager.
2. **`read_page(id, out)` / `write_page(id, in)`** move exactly one page to or from byte offset `id * kPageSize`. `read_page` self-checks that the page's stored id matches what was asked (or that it is an unformatted page). `write_page` refuses to write a page whose header id does not match its destination.
3. **`sync()`** forces buffered writes to stable storage.
4. **Exact I/O.** All reads and writes go through helpers that transfer *all* requested bytes or throw — because the raw syscalls do not promise to.

That last requirement is the whole game, and it is where this unit's forensic trap lives. Build the pager first; then we spring the trap.

Hint

Use **positioned** I/O — `pread`/`pwrite` — not `lseek`+`read`/`write`. Positioned calls carry their own offset, so there is no shared file cursor to corrupt when (in a later unit) more than one thing touches the file, and each call is one syscall instead of two. Wrap them in `pread_exact` / `pwrite_exact` that loop until the full count

is transferred, retrying on `EINTR` and advancing the buffer pointer and offset by whatever partial count came back. For durability, `fsync` (or `fdatasync`) after writing — and brace yourself for what it costs. Store the meta page immediately when the page count changes so the file size and the recorded count never disagree.

The solution

First, exact-or-die I/O. This is the entire reason short writes will not silently corrupt our database — `src/io.hpp` and `src/io.cpp`:

```
#pragma once
// Exact-or-die file I/O. POSIX read/write/pread/pwrite are allowed to
// transfer fewer bytes than asked -- a fact most programs ignore and most
// programs get away with, until they don't (see the Unit 1 forensic trap).
// The engine only ever does I/O through these two functions.
// Introduced in Challenge 1.4.

#include <sys/types.h>

#include <cstddef>
#include <string>

namespace labdb {

// Open (creating if needed) a file for read/write. Throws on failure.
int open_or_create(const std::string& path);

// Read exactly n bytes at offset off, retrying on partial reads and EINTR.
// Throws if the file ends early or the read fails.
void pread_exact(int fd, void* buf, std::size_t n, off_t off);

// Write exactly n bytes at offset off, retrying on partial writes and
// EINTR. Throws if the write fails. A short write that cannot make
// progress becomes a loud error, never silent corruption.
void pwrite_exact(int fd, const void* buf, std::size_t n, off_t off);

}  // namespace labdb
```

```
#include "io.hpp"

#include <fcntl.h>
#include <unistd.h>

#include <cstdint>

#include "common.hpp"

namespace labdb {

int open_or_create(const std::string& path) {
  const int fd = ::open(path.c_str(), O_RDWR | O_CREAT | O_CLOEXEC, 0644);
  check_sys(fd >= 0, "open");
  return fd;
}

void pread_exact(int fd, void* buf, std::size_t n, off_t off) {
  auto* p = static_cast<std::uint8_t*>(buf);
  while (n > 0) {
    const ssize_t r = ::pread(fd, p, n, off);
    if (r < 0) {
      if (errno == EINTR) continue;  // interrupted before any byte moved
      check_sys(false, "pread");
    }
    check_that(r > 0, "pread hit end of file (truncated database?)");
    p += r;
    off += r;
    n -= static_cast<std::size_t>(r);
  }
}

void pwrite_exact(int fd, const void* buf, std::size_t n, off_t off) {
  const auto* p = static_cast<const std::uint8_t*>(buf);
  while (n > 0) {
    const ssize_t w = ::pwrite(fd, p, n, off);
    if (w < 0) {
      if (errno == EINTR) continue;
```

```
      check_sys(false, "pwrite");
    }
    check_that(w > 0, "pwrite wrote zero bytes and cannot make progress");
    p += w;
    off += w;
    n -= static_cast<std::size_t>(w);
  }
}

}  // namespace labdb
```

Then the pager core, `src/pager.hpp` and `src/pager.cpp`. This is the Challenge 1.4 version — open, read, write, `extend`, `sync`. Challenge 1.5 replaces both files to add the free list, so allocation here is the bare `extend()` that grows the file by one zeroed page:

```
#pragma once
// Pager -- names pages and moves them between memory and one database file.
// Challenge 1.4 version: open/read/write/extend/sync. Allocation with a
// free list arrives in Challenge 1.5.
//
// File layout: page i lives at byte offset i * kPageSize. Page 0 is the
// meta page: magic string, page count. It never leaves the pager.
//
// Honesty note: the pager is NOT crash-safe. A crash between two related
// page writes leaves the file inconsistent, and even a single page write
// can tear. Making that survivable is the entire subject of Unit 13.
// Today's contract: correct under clean operation, loud under I/O failure.

#include <cstdint>
#include <string>

#include "page.hpp"

namespace labdb {

class Pager {
 public:
  // Opens the database file, creating and formatting it if it is empty.
  explicit Pager(const std::string& path);
  ~Pager();

  Pager(const Pager&) = delete;
  Pager& operator=(const Pager&) = delete;

  // Read/write one full page. Page 0 is off limits; ids must be in range.
  // write_page requires the page's own header id to match its destination.
  void read_page(PageId id, Page& out);
  void write_page(PageId id, const Page& in);

  // Grow the file by one zeroed page and return its id.
  PageId extend();

  // Flush the meta page and fsync the file.
  void sync();

  std::uint32_t page_count() const { return page_count_; }

 private:
  void store_meta();

  int fd_ = -1;
  std::uint32_t page_count_ = 0;
};

}  // namespace labdb
```

```
#include "pager.hpp"

#include <sys/stat.h>
#include <unistd.h>

#include <cstring>

#include "io.hpp"

namespace labdb {
```

```
namespace {

// Meta page payload, byte-exact (page 0, after the standard 20-byte header):
//
//   offset  size  field
//   ------  ----  -----------------------------
//       20     8  magic: "labdb001"
//       28     4  u32 page_count (includes page 0)
//       32     4  u32 reserved (zero; Challenge 1.5 will claim it)
//
constexpr char kMagic[8] = {'l', 'a', 'b', 'd', 'b', '0', '0', '1'};
constexpr std::size_t kMetaMagicOff = kPageHeaderSize;
constexpr std::size_t kMetaPageCountOff = kMetaMagicOff + sizeof kMagic;

off_t page_offset(PageId id) {
  return static_cast<off_t>(id) * static_cast<off_t>(kPageSize);
}

}  // namespace

Pager::Pager(const std::string& path) {
  fd_ = open_or_create(path);

  struct stat st{};
  check_sys(::fstat(fd_, &st) == 0, "fstat");

  if (st.st_size == 0) {
    // Fresh database: page 0 is born here.
    page_count_ = 1;
    store_meta();
    check_sys(::fsync(fd_) == 0, "fsync");
    return;
  }

  check_that(st.st_size % static_cast<off_t>(kPageSize) == 0,
             "database size is not a multiple of the page size");

  Page meta;
  pread_exact(fd_, meta.data(), kPageSize, 0);
  check_that(std::memcmp(meta.data() + kMetaMagicOff, kMagic,
                         sizeof kMagic) == 0,
             "bad magic: this is not a labdb file");
  page_count_ = load_u32(meta.data() + kMetaPageCountOff);
  check_that(page_count_ ==
                 static_cast<std::uint32_t>(st.st_size / kPageSize),
             "meta page count disagrees with the file size");
}

Pager::~Pager() {
  if (fd_ >= 0) {
    ::fsync(fd_);  // best effort; sync() is the checked API
    ::close(fd_);
  }
}

void Pager::read_page(PageId id, Page& out) {
  check_that(id != 0 && id < page_count_, "read_page: page id out of range");
  pread_exact(fd_, out.data(), kPageSize, page_offset(id));
  // Self-check: a formatted page must know its own name. Catches offset
  // arithmetic bugs the moment they happen instead of three units later.
  check_that(out.id() == id || out.type() == PageType::kInvalid,
             "read_page: page header claims a different id (corruption?)");
}

void Pager::write_page(PageId id, const Page& in) {
  check_that(id != 0 && id < page_count_, "write_page: page id out of range");
  check_that(in.id() == id,
             "write_page: page header id does not match destination");
  pwrite_exact(fd_, in.data(), kPageSize, page_offset(id));
}

PageId Pager::extend() {
  const PageId id = page_count_;
  Page zero;
  pwrite_exact(fd_, zero.data(), kPageSize, page_offset(id));
  ++page_count_;
  store_meta();
  return id;
}
```

```
void Pager::sync() {
  store_meta();
  check_sys(::fsync(fd_) == 0, "fsync");
}

void Pager::store_meta() {
  Page meta;
  meta.set_id(0);
  meta.set_type(PageType::kMeta);
  std::memcpy(meta.data() + kMetaMagicOff, kMagic, sizeof kMagic);
  store_u32(meta.data() + kMetaPageCountOff, page_count_);
  pwrite_exact(fd_, meta.data(), kPageSize, 0);
}

}  // namespace labdb
```

Why it works

Positioned I/O, exact-or-die. `pread`/`pwrite` take the offset as an argument, so there is no file cursor to share or corrupt and each transfer is a single syscall. Wrapping them so they loop until every byte moves turns the kernel's "may transfer fewer bytes than requested" from a lurking corruption into a non-event — on the read side. On the write side, as the trap below shows, exactness lets us *detect* a short write, which is a strictly weaker and more honest guarantee than preventing one.

Self-describing pages catch offset bugs early. `read_page` checks that the bytes at offset `id * 4096` claim to *be* page `id`; `write_page` refuses a page whose header id disagrees with its destination. Offset arithmetic is the single most bug-prone thing a pager does, and these two checks convert "wrong page silently returned, corruption discovered three units later" into "loud failure at the exact call site." The 20 bytes of header from Challenge 1.1 start paying rent here.

Immediate meta store keeps size and count honest. `extend()` grows the file and rewrites the meta page's count in the same call, so the recorded page count and the actual file size never disagree — a mismatch we treat as corruption on open. This is deliberately *not* crash-safe yet: two writes (the new page, the meta page) are not atomic together, and a crash between them leaves an inconsistency. Making that survivable is Unit 13's entire job; here we are honest that the guarantee is "correct under clean operation," nothing stronger.

The measurement

Before the trap, one measurement that will shape the rest of the book: what does durability cost? `tests/u01_fsync_bench.cpp` writes the same 4 KiB page three ways — `pwrite` alone, `pwrite`+`fdatasync`, `pwrite`+`fsync` — and times each:

```
// Challenge 1.4 measurement: what a page write costs without and with a
// durability barrier. This one table foreshadows the design of Unit 13.
#include <fcntl.h>
#include <unistd.h>

#include <cstdio>
#include <string>

#include "common.hpp"
#include "harness.hpp"
#include "io.hpp"
#include "page.hpp"

using namespace labdb;
using labdb::test::now_s;

int main() {
  const std::string path = "u01_fsync_bench.db";
  ::unlink(path.c_str());
  const int fd = open_or_create(path);

  Page page;
  page.set_id(1);
  page.set_type(PageType::kSlotted);
  // Land the page once so the loop below overwrites in place.
  pwrite_exact(fd, page.data(), kPageSize, 0);
  check_sys(::fsync(fd) == 0, "fsync");
```

```
  constexpr int kPlain = 20000, kSync = 500;

  double t0 = now_s();
  for (int i = 0; i < kPlain; ++i) pwrite_exact(fd, page.data(), kPageSize, 0);
  const double plain = (now_s() - t0) / kPlain;

  t0 = now_s();
  for (int i = 0; i < kSync; ++i) {
    pwrite_exact(fd, page.data(), kPageSize, 0);
    check_sys(::fdatasync(fd) == 0, "fdatasync");
  }
  const double fdatasync_each = (now_s() - t0) / kSync;

  t0 = now_s();
  for (int i = 0; i < kSync; ++i) {
    pwrite_exact(fd, page.data(), kPageSize, 0);
    check_sys(::fsync(fd) == 0, "fsync");
  }
  const double fsync_each = (now_s() - t0) / kSync;

  std::printf("| one 4 KiB page write | ops | us per op | vs plain |\n");
  std::printf("|---|---:|---:|---:|\n");
  std::printf("| pwrite only | %d | %.1f | 1.0x |\n", kPlain, plain * 1e6);
  std::printf("| pwrite + fdatasync | %d | %.1f | %.0fx |\n", kSync,
              fdatasync_each * 1e6, fdatasync_each / plain);
  std::printf("| pwrite + fsync | %d | %.1f | %.0fx |\n", kSync,
              fsync_each * 1e6, fsync_each / plain);

  ::close(fd);
  ::unlink(path.c_str());
  return 0;
}
```

```
$ ./bin/u01_fsync_bench

| one 4 KiB page write | ops | us per op | vs plain |
|---|---:|---:|---:|
| pwrite only | 20000 | 0.5 | 1.0x |
| pwrite + fdatasync | 500 | 552.1 | 1030x |
| pwrite + fsync | 500 | 579.6 | 1081x |
```

A durability barrier costs about **a thousand times** more than the write it makes durable. On the reference machine a bare `pwrite` into the page cache returns in half a microsecond; forcing that page to stable storage takes over half a *milli*second. (These are virtualized-host numbers; the absolute figures depend heavily on hardware, but the *ratio* — three orders of magnitude — is a property of the storage hierarchy, not the machine, and it is remarkably stable across real devices.)

Sit with that number, because it dictates architecture. If every record insert forced an `fsync`, the engine would be capped near a couple thousand writes per second no matter how fast the CPU. This single ratio is *why* databases batch writes, *why* they use a write-ahead log to turn many random page syncs into one sequential one, and *why* group commit exists. We will not solve it in Unit 1 — we will feel it here and pay it down in Unit 13. For now the pager exposes `sync()` and lets the caller decide when durability is worth a thousand-fold cost.

That is the pager working under clean conditions and its costs measured. Now we deliberately break it, because the most important thing in this unit is not that the pager works — it is understanding precisely how it fails.

The forensic trap: the torn write

Every unit in this book contains exactly one deliberate trap: a bug presented as a crime scene — symptom first, then evidence, then the diagnosis, the fix, and a regression test that locks the fix in place. The traps are the reason the book is called *Through Challenges* and not *Through Tutorials*. Here is the first.

The symptom. You are building the pager. Your `write_page` looks reasonable — it calls `pwrite` and moves on:

```
// The tempting version. It is wrong, and the wrongness is invisible
// until the worst possible moment.
void write_page(PageId id, const Page& in) {
  ::pwrite(fd_, in.data(), kPageSize, page_offset(id));  // return value ignored
}
```

Your tests pass. Every page you write reads back correctly, a thousand times in a row. You ship it. Then one day — a full disk, a process killed mid-write, a quota hit, a signal at the wrong instant — a database file comes back with *half a page* updated: the first 2 KiB is the new record, the trailing 2 KiB is whatever was there before. The page's checksum, if it had one, would be nonsense. The B+Tree node it holds points into garbage. And nothing in your code ever reported an error, because as far as your code knew, the write succeeded.

The evidence. We can reproduce this on demand, with no flaky timing and no real disk failure, by making the kernel itself perform a short write. `tools/u01_torn_write_demo.cpp` writes a full page of `0x11`, fsyncs it, then caps the file size mid-page with `RLIMIT_FSIZE` and writes a full page of `0x22` the naive way — one `pwrite`, return value dropped:

```
// THE BUG, preserved for study. This program writes a page the way naive
// storage code does: one pwrite, return value ignored. To make the
// kernel's "write() may transfer fewer bytes" clause fire on demand, it
// caps its own file size with RLIMIT_FSIZE first -- the short write below
// is performed by the kernel, not simulated.
//
// usage: u01_torn_write_demo <path>   (then hexdump the file)
#include <fcntl.h>
#include <signal.h>
#include <sys/resource.h>
#include <unistd.h>

#include <cstdio>
#include <cstring>

int main(int argc, char** argv) {
  if (argc != 2) {
    std::fprintf(stderr, "usage: %s <path>\n", argv[0]);
    return 2;
  }

  const int fd = ::open(argv[1], O_RDWR | O_CREAT | O_TRUNC, 0644);
  if (fd < 0) {
    std::perror("open");
    return 1;
  }

  unsigned char old_page[4096], new_page[4096];
  std::memset(old_page, 0x11, sizeof old_page);  // version 1 of the page
  std::memset(new_page, 0x22, sizeof new_page);  // version 2

  // Version 1 lands on disk, fully and durably.
  if (::pwrite(fd, old_page, sizeof old_page, 0) != 4096 || ::fsync(fd) != 0) {
    std::perror("setup write");
    return 1;
  }
  std::printf("setup: page written with 0x11 and fsynced\n");

  // The trap: from here on, the kernel will not let this file grow past
  // byte 2048. A write crossing that line is cut short -- the same
  // behavior POSIX permits for signals, disk-full, and quota.
  ::signal(SIGXFSZ, SIG_IGN);
  struct rlimit rl = {2048, 2048};
  ::setrlimit(RLIMIT_FSIZE, &rl);

  // ---- the buggy write: return value dropped on the floor ----
  ssize_t n = ::pwrite(fd, new_page, sizeof new_page, 0);
  std::printf("pwrite asked to write 4096 bytes... returned %zd\n", n);
  std::printf("buggy pager: ignores that number, reports the page as written\n");

  ::close(fd);
  return 0;
}
```

Run it, then look at the actual bytes on disk:

```
$ ./bin/u01_torn_write_demo torn.img
setup: page written with 0x11 and fsynced
pwrite asked to write 4096 bytes... returned 2048
buggy pager: ignores that number, reports the page as written

$ od -A d -t x1 torn.img
0000000 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22 22
*
0002048 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
*
0004096
```

There is the crime scene, in hexadecimal. Bytes 0 through 2,047 are the new page (`0x22`); bytes 2,048 through 4,095 are the *old* page (`0x11`); the tear is exactly at the 2,048-byte boundary the kernel enforced. `pwrite` told us the truth — it returned 2048, not 4096 — and the buggy pager threw that truth away. This is not a simulation of corruption; it is the real thing, produced by a real short write from a real kernel. The database file is now internally inconsistent and nothing flagged it.

The diagnosis. The bug is a wrong mental model of `write`. Programmers read `pwrite(fd, buf, 4096, off)` as "write 4,096 bytes," but POSIX only promises to write *up to* 4,096 and to return how many it actually wrote. A short write is legal and happens for mundane reasons: a signal interrupts the call, the disk fills, a quota or file-size limit is hit, the write straddles a boundary the kernel handles piecewise. Code that ignores the return value is not "usually fine" — it is *latently broken*, correct only until the first short write, and the first short write tends to arrive in production under load, which is precisely when corruption hurts most.

The fix. Never call the raw syscall from engine code. Route every transfer through `pwrite_exact`, which loops until all bytes are written and throws if it ever cannot make progress:

```
void pwrite_exact(int fd, const void* buf, std::size_t n, off_t off) {
  const auto* p = static_cast<const std::uint8_t*>(buf);
  while (n > 0) {
    const ssize_t w = ::pwrite(fd, p, n, off);
    if (w < 0) { if (errno == EINTR) continue; check_sys(false, "pwrite"); }
    check_that(w > 0, "pwrite wrote zero bytes and cannot make progress");
    p += w; off += w; n -= static_cast<std::size_t>(w);
  }
}
```

Read carefully what this does and does not buy. It converts a *silent* torn write into a *loud* exception: the process now dies with an error instead of continuing over a corrupted file. It does **not** make the write atomic — the 2 KiB that reached the disk is still there. Detection, not prevention. That distinction is the honest boundary of Unit 1: we can now *notice* every short write, but *surviving* one — so the database can be repaired to a consistent state after a crash mid-write — requires a write-ahead log, and that is the entire subject of Unit 13. A trap you can detect is the prerequisite for a trap you can survive.

The regression test. A fix without a test rots. `tests/u01_torn_write_test.cpp` rigs the same `RLIMIT_FSIZE` trap and asserts that `pwrite_exact` raises instead of returning quietly:

```
// Forensic-trap regression test: under an induced short-write condition
// (RLIMIT_FSIZE), pwrite_exact must raise a loud error -- never report
// success for a page it could not fully write.
#include <fcntl.h>
#include <signal.h>
#include <sys/resource.h>
#include <unistd.h>

#include <cstdio>
#include <cstring>
#include <exception>
#include <string>

#include "common.hpp"
#include "harness.hpp"
#include "io.hpp"
```

```
using namespace labdb;

int main() {
  const std::string path = "u01_torn_write_test.img";
  ::unlink(path.c_str());
  const int fd = open_or_create(path);

  std::uint8_t old_page[kPageSize], new_page[kPageSize];
  std::memset(old_page, 0x11, sizeof old_page);
  std::memset(new_page, 0x22, sizeof new_page);
  pwrite_exact(fd, old_page, kPageSize, 0);
  check_sys(::fsync(fd) == 0, "fsync");

  // Rig the trap: cap the file size mid-page. The kernel itself will now
  // perform a genuine short write. Ignore SIGXFSZ so we observe the return
  // value instead of dying.
  ::signal(SIGXFSZ, SIG_IGN);
  struct rlimit rl{2048, 2048};
  check_sys(::setrlimit(RLIMIT_FSIZE, &rl) == 0, "setrlimit");

  bool caught = false;
  try {
    pwrite_exact(fd, new_page, kPageSize, 0);
  } catch (const std::exception& e) {
    caught = true;
    std::printf("pwrite_exact raised as required: %s\n", e.what());
  }
  REQUIRE(caught);

  std::printf(
      "u01_torn_write_test: PASS (short write became a loud error, not "
      "silent corruption)\n");
  ::close(fd);
  ::unlink(path.c_str());
  return 0;
}
```

```
$ ./bin/u01_torn_write_test
pwrite_exact raised as required: pwrite: File too large
u01_torn_write_test: PASS (short write became a loud error, not silent corruption)
```

The trap is now permanently armed against regression: if anyone ever "simplifies" `pwrite_exact` back into a bare `pwrite`, this test fails immediately and loudly. That is the pattern every forensic trap in the book follows — reproduce the failure deterministically, fix it, and leave behind a test that makes the failure impossible to reintroduce by accident.

Files after this challenge: `src/io.hpp` (27 lines), `src/io.cpp` (48 lines), `src/pager.hpp` and `src/pager.cpp` (extended next challenge).

Challenge 1.5 — Allocation and the free list

The goal

Let the database reuse space. Add page allocation that hands out a fresh page id on demand and, crucially, *recycles* pages that have been freed instead of growing the file forever. A database that only ever grows is a database with a disk-space leak; this challenge closes it.

The challenge

Replace `src/pager.hpp` and `src/pager.cpp` to add:

- `allocate_page() -> PageId` — return a page for the caller to format. If any freed pages exist, reuse one; otherwise grow the file by one zeroed page.
- `free_page(id)` — return a page to the pool, detecting double-frees.
- `freelist_head()` and `freelist_length()` for inspection and tests.

The design question is where to keep the list of free pages. It must itself survive a restart, and it must not require a separate allocation to store — a free list that needs to allocate memory to remember free memory is a puzzle with no base case.

Hint

Store the free list *in the free pages themselves*. This is an **intrusive list**: a freed page is marked `PageType::kFree` and its `next_page` header field (reserved back in Challenge 1.1) points to the previous free-list head. The head id lives in the meta page. `free_page` pushes: mark the page free, point its `next` at the current head, make it the new head. `allocate` pops: read the head page, take its `next` as the new head, hand the id back. It is a LIFO stack threaded through the free pages — zero extra storage, and it persists because the pages persist. To catch double-frees, read the page's current type before freeing; a page already marked `kFree` is a double-free, and we throw.

The solution

The final `src/pager.hpp` and `src/pager.cpp`, now with the free list. The meta page gains a free-list-head field; `allocate_page` reuses before it extends; `free_page` refuses to free an already-free page:

```
#pragma once
// Pager -- names pages and moves them between memory and one database file.
// Challenge 1.4 (open/read/write/sync) and 1.5 (allocate/free + free list).
//
// File layout: page i lives at byte offset i * kPageSize. Page 0 is the
// meta page: magic string, page count, free-list head. It never leaves
// the pager.
//
// Honesty note, repeated wherever it matters: the pager is NOT crash-safe.
// A crash between two related page writes leaves the file inconsistent,
// and even a single page write can tear. Making that survivable is the
// entire subject of Unit 13. Today's contract is: correct under clean
// operation, loud under I/O failure.

#include <cstdint>
#include <string>

#include "page.hpp"

namespace labdb {

class Pager {
 public:
  // Opens the database file, creating and formatting it if it is empty.
  explicit Pager(const std::string& path);
  ~Pager();

  Pager(const Pager&) = delete;
  Pager& operator=(const Pager&) = delete;

  // Read/write one full page. Page 0 is off limits; ids must be in range.
  // write_page requires the page's own header id to match its destination.
  void read_page(PageId id, Page& out);
  void write_page(PageId id, const Page& in);

  // Hand out a page id: reuse the free-list head if there is one, else
  // grow the file by one zeroed page. The page's old contents are
  // unspecified -- the caller must format it before use.
  PageId allocate_page();

  // Return a page to the free list. Detects double-frees.
  void free_page(PageId id);

  // Flush the meta page and fsync the file: everything written so far is
  // now on stable storage (as far as fsync's promise goes -- Unit 13 has
  // much more to say about that sentence).
  void sync();

  std::uint32_t page_count() const { return page_count_; }
  PageId freelist_head() const { return freelist_head_; }

  // Walks the free list and counts it. O(free pages) reads; for tests
  // and the stats output, not for hot paths.
  std::uint32_t freelist_length();

 private:
  void store_meta();

  int fd_ = -1;
  std::uint32_t page_count_ = 0;
  PageId freelist_head_ = kNullPage;
```

```
};

}  // namespace labdb
```

```
#include "pager.hpp"

#include <sys/stat.h>
#include <unistd.h>

#include <cstring>

#include "io.hpp"

namespace labdb {

namespace {

// Meta page payload, byte-exact (page 0, after the standard 20-byte header):
//
//   offset  size  field
//   ------  ----  ------------------------------
//       20     8  magic: "labdb001"
//       28     4  u32 page_count (includes page 0)
//       32     4  u32 free-list head page id (0 = empty list)
//
constexpr char kMagic[8] = {'l', 'a', 'b', 'd', 'b', '0', '0', '1'};
constexpr std::size_t kMetaMagicOff = kPageHeaderSize;
constexpr std::size_t kMetaPageCountOff = kMetaMagicOff + sizeof kMagic;
constexpr std::size_t kMetaFreelistOff = kMetaPageCountOff + 4;

off_t page_offset(PageId id) {
  return static_cast<off_t>(id) * static_cast<off_t>(kPageSize);
}

}  // namespace

Pager::Pager(const std::string& path) {
  fd_ = open_or_create(path);

  struct stat st{};
  check_sys(::fstat(fd_, &st) == 0, "fstat");

  if (st.st_size == 0) {
    // Fresh database: page 0 is born here.
    page_count_ = 1;
    freelist_head_ = kNullPage;
    store_meta();
    check_sys(::fsync(fd_) == 0, "fsync");
    return;
  }

  check_that(st.st_size % static_cast<off_t>(kPageSize) == 0,
             "database size is not a multiple of the page size");

  Page meta;
  pread_exact(fd_, meta.data(), kPageSize, 0);
  check_that(std::memcmp(meta.data() + kMetaMagicOff, kMagic,
                         sizeof kMagic) == 0,
             "bad magic: this is not a labdb file");
  page_count_ = load_u32(meta.data() + kMetaPageCountOff);
  freelist_head_ = load_u32(meta.data() + kMetaFreelistOff);
  check_that(page_count_ ==
                 static_cast<std::uint32_t>(st.st_size / kPageSize),
             "meta page count disagrees with the file size");
}

Pager::~Pager() {
  if (fd_ >= 0) {
    ::fsync(fd_);  // best effort; sync() is the checked API
    ::close(fd_);
  }
}

void Pager::read_page(PageId id, Page& out) {
  check_that(id != 0 && id < page_count_, "read_page: page id out of range");
  pread_exact(fd_, out.data(), kPageSize, page_offset(id));
  // Self-check: a formatted page must know its own name. Catches offset
  // arithmetic bugs the moment they happen instead of three units later.
```

```
  check_that(out.id() == id || out.type() == PageType::kInvalid,
             "read_page: page header claims a different id (corruption?)");
}

void Pager::write_page(PageId id, const Page& in) {
  check_that(id != 0 && id < page_count_, "write_page: page id out of range");
  check_that(in.id() == id,
             "write_page: page header id does not match destination");
  pwrite_exact(fd_, in.data(), kPageSize, page_offset(id));
}

PageId Pager::allocate_page() {
  if (freelist_head_ != kNullPage) {
    const PageId id = freelist_head_;
    Page p;
    read_page(id, p);
    check_that(p.type() == PageType::kFree,
               "free-list contains a page not marked free");
    freelist_head_ = p.next_page();
    store_meta();
    return id;
  }
  // Free list empty: grow the file by one zeroed page so the file size
  // and the meta page never disagree.
  const PageId id = page_count_;
  Page zero;
  pwrite_exact(fd_, zero.data(), kPageSize, page_offset(id));
  ++page_count_;
  store_meta();
  return id;
}

void Pager::free_page(PageId id) {
  check_that(id != 0 && id < page_count_, "free_page: page id out of range");
  {
    Page current;
    read_page(id, current);
    check_that(current.type() != PageType::kFree,
               "free_page: double free (page is already on the free list)");
  }
  Page p;
  p.set_id(id);
  p.set_type(PageType::kFree);
  p.set_next_page(freelist_head_);
  write_page(id, p);
  freelist_head_ = id;
  store_meta();
}

void Pager::sync() {
  store_meta();
  check_sys(::fsync(fd_) == 0, "fsync");
}

std::uint32_t Pager::freelist_length() {
  std::uint32_t n = 0;
  PageId id = freelist_head_;
  Page p;
  while (id != kNullPage) {
    ++n;
    check_that(n <= page_count_, "free list contains a cycle");
    read_page(id, p);
    check_that(p.type() == PageType::kFree,
               "free-list contains a page not marked free");
    id = p.next_page();
  }
  return n;
}

void Pager::store_meta() {
  Page meta;
  meta.set_id(0);
  meta.set_type(PageType::kMeta);
  std::memcpy(meta.data() + kMetaMagicOff, kMagic, sizeof kMagic);
  store_u32(meta.data() + kMetaPageCountOff, page_count_);
  store_u32(meta.data() + kMetaFreelistOff, freelist_head_);
  pwrite_exact(fd_, meta.data(), kPageSize, 0);
}

}  // namespace labdb
```

Why it works

The list costs no storage because it lives in the freed pages. A page on the free list is not holding data — that is what "free" means — so its bytes are available to store the one thing we need: the id of the next free page. Threading the list through the `next_page` header field means the free list has exactly zero memory overhead and, because the pages are on disk, it survives a restart for free. The meta page holds only the head id; everything else is in the chain. This is the base case the naive design lacked: we never allocate memory to track free memory.

LIFO is deliberate, and it favors locality. `free_page` pushes onto the head and `allocate_page` pops from it, so the most recently freed page is the first reused. That tends to hand back a page whose neighbors are still warm in the page cache, and it keeps the reused region of the file compact rather than scattering writes across the whole file. A FIFO queue would need a tail pointer in the meta page and would spread reuse across older, colder pages — more bookkeeping for worse locality.

Double-free detection is nearly free and catches a real bug class. Before freeing, we read the page and check it is not already `kFree`. Freeing a page twice would create a *cycle* in the free list — the page would point at itself or at an ancestor — and a cycle turns the next allocation walk into an infinite loop or, worse, hands the same page out twice and silently aliases two logical pages onto one physical page. One type check on the way in converts that catastrophe into an immediate, loud error. (`freelist_length` also carries a cycle guard, so even a corrupted on-disk list cannot hang the counter.)

The honest caveat, again. Each `allocate` or `free` writes the meta page immediately so the on-disk head never disagrees with the pages — but, as in 1.4, the sequence of writes is not atomic. A crash partway through leaves the free list inconsistent, and repairing it belongs to Unit 13. Correct under clean operation; loud under I/O failure; not yet crash-safe.

The measurement

The claim is that allocation recycles rather than grows, and that the free list persists across a reopen. `tests/u01_pager_test.cpp` allocates five pages, writes a marker into each, closes and reopens the file to prove persistence, frees two pages, verifies the file does *not* shrink, provokes a double-free, then reopens again and shows that allocation reuses the freed pages (in LIFO order) before extending the file:

```
// Challenges 1.4 + 1.5 correctness: persistence across reopen, allocation,
// LIFO free-list reuse, double-free detection, and file-size accounting.
#include <sys/stat.h>
#include <unistd.h>

#include <cstdint>
#include <cstdio>
#include <cstring>
#include <exception>
#include <string>
#include <vector>

#include "harness.hpp"
#include "page.hpp"
#include "pager.hpp"
#include "slotted_page.hpp"

using namespace labdb;

namespace {

off_t file_size(const std::string& path) {
  struct stat st{};
  REQUIRE(::stat(path.c_str(), &st) == 0);
  return st.st_size;
}

std::vector<std::uint8_t> marker(std::uint32_t id) {
  std::vector<std::uint8_t> r(32);
  for (std::size_t i = 0; i < r.size(); ++i)
    r[i] = static_cast<std::uint8_t>(id * 131 + i * 7);
  return r;
}

}  // namespace
```

```
int main() {
  const std::string path = "u01_pager_test.db";
  ::unlink(path.c_str());

  // ---- create, allocate, write ----
  {
    Pager pager(path);
    REQUIRE(pager.page_count() == 1);  // just the meta page
    REQUIRE(file_size(path) == 4096);

    for (PageId want = 1; want <= 5; ++want) {
      const PageId id = pager.allocate_page();
      REQUIRE(id == want);  // fresh file: sequential ids
      Page page;
      SlottedPage view(page);
      view.init(id);
      auto m = marker(id);
      REQUIRE(view.insert(m).has_value());
      pager.write_page(id, page);
    }
    REQUIRE(pager.page_count() == 6);
    REQUIRE(file_size(path) == 6 * 4096);
    pager.sync();
  }  // destructor closes the file
  std::printf("PASS create+allocate: 5 data pages, file is %lld bytes\n",
              static_cast<long long>(file_size(path)));

  // ---- reopen: contents survived ----
  {
    Pager pager(path);
    REQUIRE(pager.page_count() == 6);
    for (PageId id = 1; id <= 5; ++id) {
      Page page;
      pager.read_page(id, page);
      REQUIRE(page.id() == id);
      REQUIRE(page.type() == PageType::kSlotted);
      SlottedPage view(page);
      auto got = view.read(0);
      auto expect = marker(id);
      REQUIRE(got && got->size() == expect.size());
      REQUIRE(std::memcmp(got->data(), expect.data(), expect.size()) == 0);
    }
    std::printf("PASS reopen: all 5 pages read back intact\n");

    // ---- free two pages: the file does NOT shrink ----
    pager.free_page(2);
    pager.free_page(4);
    REQUIRE(pager.freelist_head() == 4);  // LIFO
    REQUIRE(pager.freelist_length() == 2);
    REQUIRE(file_size(path) == 6 * 4096);
    std::printf("PASS free: free list = [4 -> 2], file still %lld bytes\n",
                static_cast<long long>(file_size(path)));

    // ---- double free is caught loudly ----
    bool caught = false;
    try {
      pager.free_page(4);
    } catch (const std::exception& e) {
      caught = true;
      std::printf("PASS double-free detected: %s\n", e.what());
    }
    REQUIRE(caught);
  }

  // ---- reopen again: the free list itself is durable ----
  {
    Pager pager(path);
    REQUIRE(pager.freelist_length() == 2);
    REQUIRE(pager.allocate_page() == 4);  // LIFO reuse, no growth
    REQUIRE(pager.allocate_page() == 2);
    REQUIRE(file_size(path) == 6 * 4096);
    REQUIRE(pager.allocate_page() == 6);  // list empty: extend
    REQUIRE(file_size(path) == 7 * 4096);
    std::printf(
        "PASS reuse: allocations returned 4, 2 (recycled), then 6 (extend)\n");
    std::printf("PASS file size: 24576 B during reuse, 28672 B after extend\n");
  }

  ::unlink(path.c_str());
  std::printf("u01_pager_test: PASS\n");
```

```
  return 0;
}
```

```
$ ./bin/u01_pager_test

PASS create+allocate: 5 data pages, file is 24576 bytes
PASS reopen: all 5 pages read back intact
PASS free: free list = [4 -> 2], file still 24576 bytes
PASS double-free detected: invariant violated: free_page: double free (page is already on the free list)
PASS reuse: allocations returned 4, 2 (recycled), then 6 (extend)
PASS file size: 24576 B during reuse, 28672 B after extend
u01_pager_test: PASS
```

Every claim discharged in one run: five pages persist across a close and reopen with their markers intact; freeing two pages leaves the file at 24,576 bytes rather than shrinking it (freed space is recycled, not returned to the OS); a double-free is caught with a loud invariant violation; and after a second reopen the allocator hands back pages 4 then 2 — LIFO, the reverse of the free order — before extending the file to 28,672 bytes only when the free list is empty. The free list is durable, the file grows only when it truly must, and the same self-checks from 1.4 guard every page read along the way.

Files after this challenge: `src/pager.hpp` (64 lines), `src/pager.cpp` (154 lines). The engine's storage layer is now complete: pages that hold records, survive churn, live on disk, and recycle their own space.

Challenge 1.6 — The experiment: why pages, measured

The goal

We have spent five challenges asserting that pages are the right foundation. Now prove it. Build the same million records two ways — the "obvious" design where record i lives at byte offset $i \times 64$, and our paged design where records are packed into slotted pages — and measure the difference under sequential and random access, with a cold cache and a warm one. Let the numbers, not the narrative, justify the architecture.

The challenge

Write `tests/u01_scan_bench.cpp` that:

1. Builds `u01_flat.bin` — one million 64-byte records at fixed offsets — and `u01_paged.db` — the same records through the `Pager` and `SlottedPage`.
2. Scans all million records sequentially three ways: flat with one `pread` per record, paged with one `pread` per page, and flat with one `pread` per 4 KiB chunk (the control that isolates syscall count from layout).
3. Reads 300,000 random records from each, warm and cold.
4. Drops the OS page cache between cold runs with `posix_fadvise(POSIX_FADV_DONTNEED)`, and verifies every read returns the correct record (the checksums must match, or the benchmark is measuring nothing).

Hint

The variable that dominates sequential scans is **syscalls per unit of data**, not bytes moved. A per-record read of 64 bytes still costs a full kernel round trip; amortizing that trip across 59 records in a 4 KiB page is the entire point of paging. Include the "flat in 4 KiB chunks" control so you can separate the cost of *small syscalls* from the cost of *the page structure itself* — the gap between paged and chunked flat is the price you pay for slot directories and headers, and it should be small. For random reads, expect a different story: a point lookup touches one record, so reading a whole 4 KiB page to get 64 bytes is *read amplification*, and paging should look worse, not better. Measuring the regime where your design loses is how you earn the right to claim the regime where it wins.

The solution

The full benchmark — builders, sequential scanners, random readers, cache control, and correctness checks (every scan re-sums to the known total; every random read verifies it got the record it asked

for):

```
// Challenge 1.6: the measurement pages exist for.
// One million 64-byte records, stored two ways:
//   u01_flat.bin  : record i at byte offset i*64 -- "the obvious design"
//   u01_paged.db  : the same records packed into labdb slotted pages
// Read them back sequentially and randomly, cold cache and warm, and see
// what a syscall per record actually costs.
//
// Cold cache = posix_fadvise(POSIX_FADV_DONTNEED) on the file right before
// the run: the kernel evicts the file's pages, so reads go to the device.
#include <fcntl.h>
#include <unistd.h>

#include <cstdint>
#include <cstdio>
#include <random>
#include <string>
#include <vector>

#include "common.hpp"
#include "harness.hpp"
#include "io.hpp"
#include "page.hpp"
#include "pager.hpp"
#include "slotted_page.hpp"

using namespace labdb;
using labdb::test::now_s;

namespace {

constexpr std::uint32_t kRecords = 1000000;
constexpr std::size_t kRecSize = 64;
constexpr std::uint64_t kExpectedSum = 499999500000ULL;  // sum of 0..999999

void fill_record(std::uint8_t* out, std::uint32_t id) {
  store_u32(out, id);
  for (std::size_t i = 4; i < kRecSize; ++i)
    out[i] = static_cast<std::uint8_t>(id * 2654435761u + i);
}

void drop_cache(int fd) {
  check_sys(::fsync(fd) == 0, "fsync");
  const int rc = ::posix_fadvise(fd, 0, 0, POSIX_FADV_DONTNEED);
  check_that(rc == 0, "posix_fadvise(DONTNEED) failed");
}

// ---- builders ---------------------------------------------------------

int build_flat(const std::string& path) {
  ::unlink(path.c_str());
  const int fd = open_or_create(path);
  std::vector<std::uint8_t> chunk(kPageSize);
  constexpr std::uint32_t per_chunk = kPageSize / kRecSize;  // 64
  off_t off = 0;
  for (std::uint32_t id = 0; id < kRecords;) {
    std::size_t used = 0;
    for (std::uint32_t j = 0; j < per_chunk && id < kRecords; ++j, ++id) {
      fill_record(chunk.data() + used, id);
      used += kRecSize;
    }
    pwrite_exact(fd, chunk.data(), used, off);
    off += static_cast<off_t>(used);
  }
  check_sys(::fsync(fd) == 0, "fsync");
  return fd;
}

// Returns records per page (needed to locate record i later).
std::uint32_t build_paged(Pager& pager) {
  Page page;
  SlottedPage view(page);
  PageId pid = pager.allocate_page();
  REQUIRE(pid == 1);
  view.init(pid);
  std::uint8_t rec[kRecSize];
  std::uint32_t per_page = 0;
  for (std::uint32_t id = 0; id < kRecords; ++id) {
```

```
    fill_record(rec, id);
    auto slot = view.insert(std::span<const std::uint8_t>(rec, kRecSize));
    if (!slot) {
      pager.write_page(pid, page);
      if (per_page == 0) per_page = view.slot_count();
      pid = pager.allocate_page();
      view.init(pid);
      slot = view.insert(std::span<const std::uint8_t>(rec, kRecSize));
      REQUIRE(slot.has_value());
    }
    // Packing is deterministic: record id lives at (1 + id/per_page,
    // id % per_page). The random reader below re-verifies this per read.
    if (per_page != 0) REQUIRE(*slot == id % per_page);
  }
  pager.write_page(pid, page);
  pager.sync();
  return per_page;
}

// ---- readers ----------------------------------------------------------

std::uint64_t scan_flat_per_record(int fd) {
  std::uint8_t buf[kRecSize];
  std::uint64_t sum = 0;
  for (std::uint32_t id = 0; id < kRecords; ++id) {
    pread_exact(fd, buf, kRecSize, static_cast<off_t>(id) * kRecSize);
    sum += load_u32(buf);
  }
  return sum;
}

std::uint64_t scan_flat_chunked(int fd) {
  std::uint8_t buf[kPageSize];
  std::uint64_t sum = 0;
  constexpr std::uint32_t per_chunk = kPageSize / kRecSize;
  constexpr std::uint32_t chunks = kRecords / per_chunk;
  for (std::uint32_t c = 0; c < chunks; ++c) {
    pread_exact(fd, buf, kPageSize, static_cast<off_t>(c) * kPageSize);
    for (std::uint32_t j = 0; j < per_chunk; ++j)
      sum += load_u32(buf + j * kRecSize);
  }
  return sum;
}

std::uint64_t scan_paged(Pager& pager) {
  Page page;
  std::uint64_t sum = 0;
  const std::uint32_t pages = pager.page_count();
  for (PageId id = 1; id < pages; ++id) {
    pager.read_page(id, page);
    SlottedPage view(page);
    const std::uint16_t n = view.slot_count();
    for (std::uint16_t s = 0; s < n; ++s) {
      auto rec = view.read(s);
      sum += load_u32(rec->data());
    }
  }
  return sum;
}

std::uint64_t random_flat(int fd, const std::vector<std::uint32_t>& ids) {
  std::uint8_t buf[kRecSize];
  std::uint64_t sum = 0;
  for (std::uint32_t id : ids) {
    pread_exact(fd, buf, kRecSize, static_cast<off_t>(id) * kRecSize);
    REQUIRE(load_u32(buf) == id);  // right record, verified
    sum += load_u32(buf);
  }
  return sum;
}

std::uint64_t random_paged(Pager& pager, std::uint32_t per_page,
                           const std::vector<std::uint32_t>& ids) {
  Page page;
  std::uint64_t sum = 0;
  for (std::uint32_t id : ids) {
    const PageId pid = 1 + id / per_page;
    const auto slot = static_cast<std::uint16_t>(id % per_page);
    pager.read_page(pid, page);
    SlottedPage view(page);
```

```
    auto rec = view.read(slot);
    REQUIRE(rec.has_value());
    REQUIRE(load_u32(rec->data()) == id);  // locator verified per read
    sum += load_u32(rec->data());
  }
  return sum;
}

}  // namespace

int main() {
  const std::string flat_path = "u01_flat.bin";
  const std::string paged_path = "u01_paged.db";

  std::printf("building %u records of %zu bytes each...\n\n", kRecords,
              kRecSize);
  double t0 = now_s();
  const int flat_fd = build_flat(flat_path);
  const double t_flat = now_s() - t0;

  ::unlink(paged_path.c_str());
  t0 = now_s();
  Pager pager(paged_path);
  const std::uint32_t per_page = build_paged(pager);
  const double t_paged = now_s() - t0;
  // Second fd on the same file, used only to evict its cache pages.
  const int paged_fd = open_or_create(paged_path);

  const std::uint32_t data_pages = pager.page_count() - 1;
  std::printf("u01_flat.bin : records at fixed 64 B offsets, %.1f MiB, built in %.2f s\n",
              double(kRecords) * kRecSize / (1024.0 * 1024.0), t_flat);
  std::printf("u01_paged.db : %u records/page, %u data pages, %.1f MiB, built in %.2f s\n\n",
              per_page, data_pages,
              double(pager.page_count()) * kPageSize / (1024.0 * 1024.0),
              t_paged);

  // ---- sequential scan of all records --------------------------------
  std::printf("sequential scan of all %u records:\n\n", kRecords);
  std::printf("| layout | read granularity | syscalls | cold (s) | cold ns/rec | warm (s) | warm ns/rec |\n");
  std::printf("|---|---|---:|---:|---:|---:|---:|\n");

  double a_warm, b_warm, c_warm;
  {
    drop_cache(flat_fd);
    t0 = now_s();
    std::uint64_t s = scan_flat_per_record(flat_fd);
    const double cold = now_s() - t0;
    REQUIRE(s == kExpectedSum);
    t0 = now_s();
    s = scan_flat_per_record(flat_fd);
    a_warm = now_s() - t0;
    REQUIRE(s == kExpectedSum);
    std::printf("| flat | 64 B, one pread per record | %u | %.3f | %.0f | %.3f | %.0f |\n",
                kRecords, cold, cold / kRecords * 1e9, a_warm,
                a_warm / kRecords * 1e9);
  }
  {
    drop_cache(paged_fd);
    t0 = now_s();
    std::uint64_t s = scan_paged(pager);
    const double cold = now_s() - t0;
    REQUIRE(s == kExpectedSum);
    t0 = now_s();
    s = scan_paged(pager);
    b_warm = now_s() - t0;
    REQUIRE(s == kExpectedSum);
    std::printf("| paged (labdb) | 4 KiB, one pread per page | %u | %.3f | %.0f | %.3f | %.0f |\n",
                data_pages, cold, cold / kRecords * 1e9, b_warm,
                b_warm / kRecords * 1e9);
  }
  {
    drop_cache(flat_fd);
    t0 = now_s();
    std::uint64_t s = scan_flat_chunked(flat_fd);
    const double cold = now_s() - t0;
    REQUIRE(s == kExpectedSum);
    t0 = now_s();
    s = scan_flat_chunked(flat_fd);
    c_warm = now_s() - t0;
    REQUIRE(s == kExpectedSum);
```

```
    std::printf("| flat | 4 KiB, one pread per chunk | %u | %.3f | %.0f | %.3f | %.0f |\n",
                kRecords / (unsigned)(kPageSize / kRecSize), cold,
                cold / kRecords * 1e9, c_warm, c_warm / kRecords * 1e9);
  }
  std::printf("\nwarm sequential, per-record syscalls vs paged: %.1fx slower\n",
              a_warm / b_warm);
  std::printf("warm sequential, paged vs raw 4 KiB chunks: %.2fx (the price of structure)\n\n",
              b_warm / c_warm);

  // ---- random point reads ------------------------------------------
  std::mt19937 rng(7);
  std::uniform_int_distribution<std::uint32_t> id_dist(0, kRecords - 1);
  std::vector<std::uint32_t> warm_ids(300000), cold_ids(30000);
  for (auto& id : warm_ids) id = id_dist(rng);
  for (auto& id : cold_ids) id = id_dist(rng);

  std::printf("random point reads (record ids drawn uniformly, fixed seed):\n\n");
  std::printf("| layout | bytes moved per read | state | reads | us per read |\n");
  std::printf("|---|---|---|---:|---:|\n");

  // Warm first: both files are fully cached from the sequential scans.
  t0 = now_s();
  std::uint64_t s = random_flat(flat_fd, warm_ids);
  const double a_rw = now_s() - t0;
  labdb::test::do_not_optimize(s);
  t0 = now_s();
  s = random_paged(pager, per_page, warm_ids);
  const double b_rw = now_s() - t0;
  labdb::test::do_not_optimize(s);

  // Cold: evict both files, then read a smaller batch.
  drop_cache(flat_fd);
  t0 = now_s();
  s = random_flat(flat_fd, cold_ids);
  const double a_rc = now_s() - t0;
  labdb::test::do_not_optimize(s);
  drop_cache(paged_fd);
  t0 = now_s();
  s = random_paged(pager, per_page, cold_ids);
  const double b_rc = now_s() - t0;
  labdb::test::do_not_optimize(s);

  std::printf("| flat | 64 B | cold | %zu | %.2f |\n", cold_ids.size(),
              a_rc / double(cold_ids.size()) * 1e6);
  std::printf("| paged (labdb) | 4096 B | cold | %zu | %.2f |\n",
              cold_ids.size(), b_rc / double(cold_ids.size()) * 1e6);
  std::printf("| flat | 64 B | warm | %zu | %.2f |\n", warm_ids.size(),
              a_rw / double(warm_ids.size()) * 1e6);
  std::printf("| paged (labdb) | 4096 B | warm | %zu | %.2f |\n",
              warm_ids.size(), b_rw / double(warm_ids.size()) * 1e6);
  std::printf("\nrandom point reads, paged vs flat, warm: %.2fx\n", b_rw / a_rw);

  // The two files are left on disk deliberately: inspect them with od,
  // filefrag, or ls -ls. `make clean` removes them.
  ::close(flat_fd);
  ::close(paged_fd);
  return 0;
}
```

The measurement

```
$ ./bin/u01_scan_bench

building 1000000 records of 64 bytes each...

u01_flat.bin : records at fixed 64 B offsets, 61.0 MiB, built in 0.31 s
u01_paged.db : 59 records/page, 16950 data pages, 66.2 MiB, built in 0.34 s

sequential scan of all 1000000 records:

| layout | read granularity | syscalls | cold (s) | cold ns/rec | warm (s) | warm ns/rec |
|---|---|---:|---:|---:|---:|---:|
| flat | 64 B, one pread per record | 1000000 | 0.447 | 447 | 0.437 | 437 |
| paged (labdb) | 4 KiB, one pread per page | 16950 | 0.031 | 31 | 0.014 | 14 |
| flat | 4 KiB, one pread per chunk | 15625 | 0.026 | 26 | 0.013 | 13 |

warm sequential, per-record syscalls vs paged: 31.4x slower
warm sequential, paged vs raw 4 KiB chunks: 1.08x (the price of structure)
```

```
random point reads (record ids drawn uniformly, fixed seed):

| layout | bytes moved per read | state | reads | us per read |
|---|---|---|---:|---:|
| flat | 64 B | cold | 30000 | 19.75 |
| paged (labdb) | 4096 B | cold | 30000 | 17.19 |
| flat | 64 B | warm | 300000 | 0.54 |
| paged (labdb) | 4096 B | warm | 300000 | 0.80 |

random point reads, paged vs flat, warm: 1.48x
```

Three findings, each worth reading deliberately.

Sequential scan: structure nearly free, small syscalls ruinous. Warm, the paged scan reads all million records at about 15 nanoseconds each; the per-record flat scan takes about 438 — a **31× difference**, bought almost entirely by making 16,950 syscalls instead of 1,000,000. The control proves it: flat read in 4 KiB chunks (15,625 syscalls) lands at ~13 ns/record, so paging is within about 1.1× of raw block I/O. That small gap — the "price of structure" — is everything the slot directory and per-page headers cost us on a scan, and it is nearly nothing. The structure that will let us do B+Trees, catalogs, and query plans is almost free on the workload databases run most.

Random point reads: paging loses, exactly as predicted. Warm, a random flat read takes ~0.54 µs and a paged read ~0.80 µs — paging is about **1.5× slower**. This is not a defect; it is read amplification stated in numbers: to return one 64-byte record the paged reader copies a whole 4,096-byte page (64× the bytes), so for pure point lookups with no locality it does more work. We show this rather than hide it because it defines what Unit 1 does *not* solve. Pages win scans; they do not win random point lookups. Two different tools fix that later — an *index* (Unit 2's B+Tree) to find a record without scanning, and a *buffer pool* (Unit 4) to keep hot pages resident so the amplified read hits RAM, not disk. Neither is a page-layout change; both are layers above it.

A note on "cold" numbers, honestly. The cold rows exist to show the *shape* of the effect, but read them with a caveat: this book's reference machine is a virtualized host, and `posix_fadvise(DONTNEED)` evicts pages from the guest's cache while the hypervisor may still be caching the file underneath. So the cold figures here understate the true disk-versus-cache gap you would see on bare metal with a spinning disk or a cold SSD — sometimes dramatically. The durable, machine-independent signals are the *warm ratios* and the *syscall counts*, which are properties of the software, not the storage: 31× for per-record versus paged, ~1.1× for the price of structure, ~1.5× read amplification on point lookups. Those ratios reproduce anywhere; the absolute microseconds do not. This is the book's measurement discipline in miniature — report what the software determines, flag what the environment contaminates.

One more look — at the physical file. The paged file is slightly larger (66.2 vs 61.0 MiB — the 8% is headers and slot directories and the last page's slack) and its bytes are laid out in a handful of extents:

```
$ filefrag u01_flat.bin u01_paged.db
u01_flat.bin: 1 extent found
u01_paged.db: 3 extents found
```

Both files are effectively contiguous on disk — a few extents for a 66 MiB file is the filesystem's normal allocation, not fragmentation, and it does not meaningfully affect the sequential numbers above. (If you run the demo yourself and see a large *cold* first-read anomaly, that is host-cache warmup on the virtualized disk, the same effect described in the note above, not the file being fragmented.) We flag it so a surprising `filefrag` line does not get over-read: the extent count is benign.

Where we landed

What we measured. Not opinions — numbers, each from a real run of code printed in this unit:

- A page holds 59 sixty-four-byte records at 92% payload utilization; the 4-byte slot is a rounding error for mid-size records and a real tax only for tiny ones (79% at 16 bytes) or a quantization

loss for large ones (75% at 1 KiB). *(Challenge 1.2)*
- `memcpy`-based field access compiles to the identical single `movzwl` instruction as an unsafe pointer cast — the safety cost is exactly zero. *(Challenge 1.1)*
- A page survives 135,000 inserts and 135,000 erases with every invariant intact; compaction costs about 0.14 ns per live byte, linear as predicted, and amortizes to ~215 ns per churn pair. *(Challenge 1.3)*
- An `fsync` costs about 1,000× a cached `pwrite` — the single ratio that dictates why write-ahead logging has to exist. *(Challenge 1.4)*
- Paged sequential scans run 31× faster than per-record I/O and within 1.1× of raw block reads; paged point lookups run 1.5× *slower* than flat, the read-amplification price we pay and will fix with an index and a buffer pool. *(Challenge 1.6)*

What we now know. The file is an array of self-describing 4 KiB pages. Records live inside pages, addressed by stable slot ids, and the page reclaims its own space under churn. The pager moves whole pages to and from disk with exact-or-die I/O, recycles freed pages through an intrusive free list at zero storage cost, and catches offset bugs and double-frees the moment they happen. And we know — because we built the crime scene — exactly how a short write tears a page, how to detect it, and why detection is not yet survival.

What still hurts. One specific, measured thing: **to find a record, we have no choice but to scan.** Every one of those fast sequential scans read *all* million records because a slotted page can tell you what is in slot 7, but nothing can tell you *which* slot — or which page — holds the row where `id = 481116` without looking at all of them. Point lookups are O(n) reads. That is not a small inefficiency; it is the difference between a database and a file. Three other bills are also outstanding, each with a named payer: the 1,000× `fsync` cost and the torn write we can detect but not survive both wait for **Unit 13's** write-ahead log; the read amplification on point lookups waits for **Unit 4's** buffer pool; and `allocate_page` currently issues three `pwrite`s (zero the page, then the meta page, then the caller's write) for one logical allocation — write amplification the buffer pool will also help retire.

But the scan problem comes first, because everything else in the engine depends on finding data quickly. The universal answer is to keep records *sorted* and *indexed* in a structure whose height grows logarithmically — so a lookup among a billion rows costs a handful of page reads instead of a billion. That structure is the B+Tree, and building its insert and search paths is **Unit 2**. The fan-out arithmetic you already met in the capacity table — how many entries fit in a page — is about to decide how tall that tree is, and therefore how many disk reads every query in the rest of this book will cost.

The engine so far

file	lines	role
`src/common.hpp`	76	page size, ids, error policy, byte-exact field access
`src/page.hpp`	71	the 4 KiB `Page` and its self-describing header
`src/slotted_page.hpp`	167	variable-length records, stable slot ids, compaction
`src/io.hpp`	27	exact-or-die I/O declarations
`src/io.cpp`	48	`pread_exact` / `pwrite_exact`
`src/pager.hpp`	64	the pager interface: read-/write/allocate/free/sync
`src/pager.cpp`	154	pager implementation, meta page, intrusive free list
engine total	**607**	**the storage layer, complete**

Supporting it: about 1,160 lines of tests, benchmarks, forensic-trap demo, and snapshots in `tests/` ↪ , `tools/`, and `snapshots/` — every one of which compiles and runs with the commands in the appendix below.

Run it yourself

Everything in this unit reproduces from the repository root with three commands. No dependencies beyond a C++20 compiler and a POSIX system; every program is built with `-std=c++20 -O2 -Wall -Wextra` and compiles without warnings.

```
$ make all      # build the engine, tests, benchmarks, and the trap demo
$ make test     # correctness: page layout, slotted churn, pager, torn-write
$ make bench    # measurements: header access, compaction, fsync, scan
```

`make test` runs the layout test, the 300,000-operation slotted-page churn test against its reference model, the pager persistence/allocation/free test, the torn-write regression test, and the two snapshot smoke tests that verify the intermediate Challenge 1.2 and 1.4 listings compile and run. `make bench` runs the four measurement programs whose tables appear above.

To see the forensic trap with your own eyes — the actual torn page in hexadecimal:

```
$ ./bin/u01_torn_write_demo torn.img
$ od -A d -t x1 torn.img      # 0x22 through byte 2047, 0x11 after: the tear
$ rm -f torn.img
```

The scan benchmark leaves `u01_flat.bin` and `u01_paged.db` on disk so you can inspect them (`ls -ls`, `filefrag`, `od`); `make clean` removes them along with the binaries. Every measurement in this unit uses a fixed random seed, so your runs will differ only in timing — never in counts, capacities, or the byte contents of that torn page. The absolute nanoseconds depend on your hardware; the ratios, as discussed, should not.

Or reproduce the whole unit end to end, trap included, with the bundled script:

```
$ sh scripts/run_unit01.sh
```

What comes next

This sample contains the book's introduction and all of **Unit 1** — slotted pages and the pager, the foundation the rest of the engine builds on without exception for the next seven units.

The complete book continues with:

Part I — Storage *(continued)*

- **Unit 2** — B+Tree: Insert & Search — logarithmic lookup, proven against a million-key tree
- **Unit 3** — B+Tree: Delete & Range Scans — rebalancing, merges, ordered range queries
- **Unit 4** — The Buffer Pool — a real cache with clock eviction, measured 2.5x faster with identical logical work

Part II — From Bytes to a Query

- **Unit 5** — The Record Layer — typed rows: a schema-aware codec and a table heap, reached by key through the index
- **Unit 6** — The Catalog — persistent schemas and named tables: the engine's self-knowledge
- **Unit 7** — The Front End — a SQL tokenizer and recursive-descent parser, with compiler-quality caret-pointed errors
- **Unit 8** — The Executor — **the milestone**: a real executor that runs `CREATE TABLE`, `INSERT`, and `SELECT` end to end, against a database of hundreds of thousands of rows

By the book's last page, the engine you started in Unit 1 answers a SQL query it parsed from text, against rows it stored on disk, through an index it built and a cache it manages itself — and every number in the book came from actually running that code.

Get the complete book

Build a SQL Database Engine in C++ — Parts I & II is available now on Leanpub.

A companion volume, continuing the engine into crash recovery, concurrency, and performance, is available separately and as a discounted bundle.