Building a Database Engine: Transactions & Concurrency

Write-Ahead Logging, Crash Recovery, Rollback, Locking, MVCC, Foreign Keys & Query Planning — From Scratch

Topics covered:

A real write-ahead log & crash recovery routine

Undo logging & rollback · locking & a real deadlock · MVCC

Multi-table foreign keys · nested-loop & hash joins

A cost-based query planner

Capstone: every subsystem wired together into one real, disk-backed, transactional, concurrent, multi-table engine — with two genuine integration bugs found and fixed along the way

Exercises: 30 hands-on exercises with worked, verified solutions

Format: A4 · Dark-theme code examples

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 10

Why Transactions? ACID, and What Course 1 Doesn't Guarantee

Building a Database Engine: Transactions & Concurrency

Chapter 1 · Why Transactions? ACID, and What Course 1 Doesn't Guarantee

Course 1 ended with a real, working single-table engine: records, pages, a heap table, B-tree indexes, and a small SQL-like query language, all genuinely wired together and verified end to end in its own capstone. What it never once claimed, anywhere, is that any of it survives going wrong *partway through*. This chapter doesn't add a single line of new engine code — it goes back to Course 1's own finished code and asks it two direct, uncomfortable questions, verifying the answers with real Python rather than assuming them.

SCOPE NOTE — READ THIS BEFORE ANYTHING ELSE

This course, Course 2, is where transactions, crash recovery, concurrent access, multi-table joins, and query planning across more than one table are actually built. Nothing in this chapter is a new feature — it's a deliberately honest audit of Course 1's own real engine, using Course 1's own real classes (`Page`, `HeapFile`, `HeapTable`, `BTreeNode`, `Database`), unmodified.

ACID, Stated Concretely

Four properties a database is expected to guarantee around any single operation (or group of operations run together as one unit, a **transaction**):

PROPERTY	WHAT IT ACTUALLY MEANS
Atomicity	A multi-step operation either fully happens, or none of it does — never left half-done
Consistency	The database moves from one genuinely valid state to another — nothing that's supposed to stay true (like "every row is reachable through its own index") is allowed to silently stop being true
Isolation	Two operations running at the same time don't see each other's own half-finished work
Durability	Once an operation is confirmed done, it survives a crash — a power loss, a killed process, a kernel panic

Course 1's own capstone chapter closed with an honest "What This Course Doesn't Cover" section naming exactly this gap — it just never demonstrated *what specifically* goes wrong without it. That's this chapter's job.

Finding 1: A Real, Verified Consistency Failure

Course 1's own `Database.insert()` is a genuine two-step operation: write the row to the heap table, then update every index registered for that table.

```
def insert(self, table_name, values):
    schema, table = self.tables[table_name]
    location = table.insert(values)                    # STEP 1: physically write the row
    column_names = [name for name, t in schema.columns]
    for (t_name, col_name), root in list(self.indexes.items()):
        if t_name == table_name:
            idx = column_names.index(col_name)
            self.indexes[(t_name, col_name)] = btree_insert(root, values[idx], location)   # STEP 2: 
    return location
```

Nothing in Course 1 guarantees these two steps happen together. To verify what a real interruption between them actually looks like, this test skips `Database.insert()` entirely for one row and calls the heap table's own `insert()` directly — exactly what a process crash between step 1 and step 2 would leave behind.

```
db.insert('accounts', [1, 'Alice', 500])   # a real, complete insert -- both steps ran

# simulate a crash: call the heap table's own insert() directly, bypassing
# Database.insert()'s own second step (the index update) entirely
schema, table = db.tables['accounts']
table.insert([2, 'Bob', 300])
```

VERIFIED DIRECTLY — THE TABLE AND ITS OWN INDEX SILENTLY DISAGREE

A real scan of the heap table finds **both** rows: `[[1, 'Alice', 500], [2, 'Bob', 300]]` — row 2 is genuinely, physically present on disk. But `db.select('accounts', where=('id', '=', 2))`, which Chapter 9's own real query planner routes through the index, returns `[]` — empty. The row exists. A query against it, using the index, reports that it doesn't. Two parts of the same database have silently drifted out of agreement with each other, because nothing enforces that a multi-step operation either fully completes or doesn't happen at all — precisely what **atomicity** is supposed to guarantee, and precisely what Course 1 never built.

This isn't a hypothetical worst case dreamed up for the sake of a scary example — it's the exact, ordinary shape every real insert already takes. Any interruption at all between its two steps — a crash, a killed process, even an unhandled exception raised partway through the index-update loop — leaves the database in exactly this state.

Finding 2: The Index Itself Never Touches Disk

A more basic question: does a B-tree index even survive an ordinary, controlled restart — closing the Python process and opening a fresh one pointed at the same files, the way any real application actually re-opens a database?

```
db_a = Database(directory)
db_a.create_table('products', [('id', 'INTEGER'), ('name', 'TEXT')])
db_a.insert('products', [1, 'Widget'])
db_a.insert('products', [2, 'Gadget'])
db_a.create_index('products', 'id')

# simulate a real restart: a FRESH Database object, pointing at the SAME directory
db_b = Database(directory)
db_b.create_table('products', [('id', 'INTEGER'), ('name', 'TEXT')])
```

VERIFIED DIRECTLY — THE DATA SURVIVES, THE INDEX DOESN'T

`list(db_b.tables['products'][1].scan())` after "restarting" returns both real rows, completely intact — Chapter 4's own `load_page` correctly reads back real bytes from a real file, exactly as it was verified to do in Course 1. But `db_b.indexes` is `{}` — empty. The entire B-tree built across Chapters 5–6 is gone. A `BTreeNode` was always a plain, in-memory Python object — nothing in this engine ever gave it a byte format, a file, or a way to be written to disk at all, unlike `Page`, which genuinely has all three.

A query for `WHERE id = 1` against `db_b` still returns the right answer here — but only because Chapter 9's own `choose_plan()` correctly sees no index registered for that column and safely falls back to a full scan. This particular case is a silent **performance** loss, not a correctness bug: every index would need to be rebuilt from scratch, by hand, after every single restart, or a real application simply never gets the benefit Course 1's own Chapter 9 built it for in the first place.

Separately, real database durability has a second, deeper layer worth naming honestly even without simulating it directly: Course 1's own `HeapFile.write_page()` calls `self.file.flush()` after every write, but never `os.fsync()`. This is a well-established, uncontroversial fact about how operating systems work, not something that needs a live crash to demonstrate — `flush()` only pushes data out of Python's own internal buffer and into the operating system's page cache; the OS itself is still free to hold that data in memory for a while before actually committing it to the physical disk. A real power loss between those two points loses data Python already reported as "written." Chapter 2 builds the real mechanism — write-ahead logging — that closes this exact gap.

What Actually Happened, Underneath Both Findings

Both findings trace back to the same root cause: Course 1 built real *mechanisms* — pages, a heap file, a B-tree, a query planner — but never built any *protocol* governing what happens when something interrupts a sequence of steps partway through. That protocol is precisely what this course exists to build:

THIS CHAPTER'S FINDING	RESOLVED IN
Data written to Python's own file buffer isn't guaranteed to survive a real power loss	Chapter 2 — Write-Ahead Logging
A crash mid-write can leave a page (or the whole table) in an unknown state	Chapter 3 — Crash Recovery: Replaying the Log
A two-step insert with only step 1 completed leaves the index silently wrong	Chapter 4 — Atomicity: Undo Logging & Rollback
Two operations running at the same time could interfere with each other (not yet tested directly — Course 1's engine has never run two operations concurrently at all)	Chapters 5–6 — Locking and MVCC

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
The index/data consistency bug	Building a Database Engine: Storage & Query Fundamentals Chapters 4 and 9 (Course 1) — the exact two real methods, `HeapTable.insert()` and `Database.insert()`, whose gap between them is demonstrated here
The in-memory-only B-tree	Building a Database Engine: Storage & Query Fundamentals Chapters 5–6 (Course 1) — the real `BTreeNode` class, confirmed here to have no on-disk representation at all
flush() vs. a genuine OS-level fsync	Technical Support: Backup & Disaster Recovery Basics — the same "silence isn't proof of success" caution, applied one layer lower, to a single write instead of a whole backup job

Hands-On Exercises

EXERCISE 1

Rebuild the index used in Finding 1 (via `db.create_index('accounts', 'id')`, run *after* the "crashed" row 2 was inserted) and re-run the query for `id = 2`. Confirm it now returns the row correctly, and explain in your own words why this is a working manual fix rather than a real guarantee.

View solution

EXERCISE 2

Construct a scenario with two indexes on the same table (e.g. on `id` and on `name`) and simulate a crash between step 1 and the FIRST of the two index updates in the loop — so both indexes are stale, not just one. Verify both queries return wrong (empty) results for the crashed row.

View solution

EXERCISE 3

Read Course 1's own `HeapFile.write_page()` method directly and confirm, by inspection, that it never calls `os.fsync()` anywhere. Then look up what `os.fsync()` actually does in Python's own documentation, and explain in one paragraph, in your own words, why `flush()` alone doesn't guarantee durability against a real power loss.

View solution

CHAPTER 1 QUICK REFERENCE

- **ACID:** Atomicity, Consistency, Isolation, Durability — four guarantees Course 1's engine never made
- **Verified Finding 1:** an interrupted two-step insert leaves the index silently wrong — the row is really on disk, but a query using the index returns nothing
- **Verified Finding 2:** the entire B-tree index lives only in memory — it vanishes completely on every restart, while the underlying table data survives intact
- **Named, not tested directly:** `write_page()` calls `flush()` but never `os.fsync()` — a real, well-documented OS-level durability gap
- **Next chapter:** Write-Ahead Logging — the real mechanism that makes a write durable before the data itself is even touched

CHAPTER 2 OF 10

Write-Ahead Logging: Durability Before the Data Itself

Building a Database Engine: Transactions & Concurrency

Chapter 2 · Write-Ahead Logging: Durability Before the Data Itself

Chapter 1 named the gap without closing it: `write_page()` calls `flush()` but never `os.fsync()`, and even if it did, a crash between logging intent and applying it could still leave a table's own pages and indexes disagreeing. A write-ahead log is the real, standard mechanism that turns "I hope this happened" into something verifiable — write down exactly what's about to change, force it to disk, and only then touch the actual data. If a crash happens after the log write, the record survives. If it happens before, nothing was promised yet. There's no third case.

A Real WAL Record Format

Each record is length-prefixed and checksummed, so a reader never has to guess where one record ends and the next begins, or trust bytes it can't verify:

```
# [4 bytes: payload length][4 bytes: CRC32 of payload][payload]
# payload = [8 bytes: LSN][4 bytes: page_num][PAGE_SIZE bytes: new page data]

def build_record(lsn, page_num, new_page_bytes):
    payload = struct.pack('>QI', lsn, page_num) + new_page_bytes
    crc = zlib.crc32(payload)
    return struct.pack('>II', len(payload), crc) + payload
```

The **LSN** (log sequence number) is a strictly increasing integer identifying each record — Chapter 3's own recovery routine will need it to know exactly where it left off. This chapter logs the entire *new* page image, not a small delta — deliberately simple, and it buys something important later: reapplying an already-applied record is harmless, since writing the same complete page twice produces the identical result either time (this matters directly for Exercise 1, below).

The One Non-Negotiable fsync in This Engine

```
class WAL:
    def append(self, page_num, new_page_bytes):
        lsn = self.next_lsn
        self.next_lsn += 1
        record = build_record(lsn, page_num, new_page_bytes)
        self.file.seek(0, os.SEEK_END)
        self.file.write(record)
        self.file.flush()
        os.fsync(self.file.fileno())      # the durability guarantee actually lives here
        return lsn

def write_page_with_wal(wal, heap_file, page_num, page):
    wal.append(page_num, bytes(page.data))   # STEP 1: log first, durably
    heap_file.write_page(page_num, page)     # STEP 2: apply to the real data file
```

WHY FSYNC SPECIFICALLY HERE, AND NOT EVERYWHERE

Chapter 1 pointed out that Course 1's own `HeapFile.write_page()` never calls `os.fsync()` at all — and this course still doesn't add it there. Calling `fsync()` on every single data-page write would be correct but needlessly slow, forcing a real disk commit on every write regardless of size. The actual guarantee a WAL needs is narrower: only the *log* write has to be durable before the operation is considered "logged" at all — the data page itself can be written lazily, exactly as it always was, because the log is now the thing standing behind it if a crash happens first.

Finding 1: A Logged-but-Unapplied Change Survives a Simulated Crash

A "crash" is simulated the same honest way Chapter 1 did it: call `wal.append()` directly and never call `heap_file.write_page()` at all — exactly what a process death immediately after the WAL's own `fsync()` returns, but before the data write runs, actually looks like.

VERIFIED DIRECTLY — THE DATA FILE IS HONESTLY UNTOUCHED, THE LOG IS HONESTLY INTACT

After the simulated crash, reopening the heap file (a fresh `HeapFile` pointed at the same path) shows the target page completely unchanged from before — the intended update genuinely never happened at the data layer, exactly as expected; nothing here replays it yet. But reopening the WAL file and calling `read_all_records()` finds **exactly one record**, with the correct LSN, correct page number, and a `new_page_bytes` value that matches the intended update *byte-for-byte*. The data update was lost. The information needed to redo it wasn't.

That distinction — the update didn't happen, but nothing about it was forgotten — is the entire point of this chapter, and precisely what Chapter 3's own recovery routine will use.

Finding 2: A Torn Trailing Record — Two Very Different Failure Modes

A real crash can interrupt the WAL's own append in the middle, too — the OS commits some of a record's bytes but not all of them before the process dies. This test builds one complete record, then writes only the first half of a second record's real bytes, simulating exactly that.

VERIFIED DIRECTLY — THE REAL READER DISCARDS THE TORN RECORD CLEANLY

`read_all_records()` (bounds-checked and checksum-verified) returns exactly **1** record — the complete first one. It correctly recognizes that the second record's declared length runs past the actual end of the file, stops reading right there, and simply treats everything after that point as if it had never been logged at all — which is honest, since it never finished being written durably in the first place.

VERIFIED DIRECTLY — A NAIVE READER IS SILENTLY, ACTIVELY WRONG, NOT JUST CARELESS

A naive reader with no bounds check and no checksum — one that simply trusts whatever length the (fully intact) 8-byte header of the torn record declares — doesn't crash and doesn't notice anything is wrong. It slices out whatever bytes happen to be available (Python slicing silently returns fewer bytes than requested rather than raising an error), successfully unpacks a genuinely correct LSN and page number from the surviving front of the payload, and reports a second, fully-formed-looking record — with a `new_page_bytes` field that is only **118 bytes** long, not the required 256. Fed into a real recovery routine, this would silently write a truncated, garbage page into the middle of a real table, under a completely correct LSN and page number that give no outward sign anything is wrong. A crash here would have been the *safe* outcome; this is worse than a crash.

Finding 2b: Why Bounds-Checking Alone Isn't Enough

A truncated file is one kind of corruption. A single flipped bit somewhere *inside* an otherwise complete, correctly-sized record is another — and a length check alone has nothing to say about it.

VERIFIED DIRECTLY — ONLY THE CHECKSUM CATCHES IN-PLACE CORRUPTION

A record was built normally, then one byte deep inside its payload (well past the LSN and page number fields) was flipped, leaving the file's own total length and the record's own declared length completely correct. A bounds-check-only reader (no checksum) reports **1** record — it accepts the corrupted payload outright, since nothing about the file's shape looks wrong. The checksum-verified reader reports **0** — it recomputes the payload's real CRC32, finds it doesn't match the stored one, and correctly rejects the record. Truncation and in-place corruption are genuinely different failure shapes, and this engine's own two independent checks — length bounds, then checksum — exist because neither one alone catches both.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Log-before-apply ordering, and a durable-but-unapplied record	Chapter 1 (this course) — the exact `flush()`-without-`fsync()` gap identified there is what this chapter's WAL closes, specifically for the log itself, not every write

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A torn trailing record correctly discarded	Chapter 3 (this course) — recovery starts by reading exactly this list of valid records and deciding what still needs to be reapplied
A naive reader silently fabricating a plausible-looking corrupted record	Building a Web Browser Engine: Layout & Rendering Chapter 8's own bounds-clipping bug, and this site's own recurring "confidently wrong is worse than a crash" theme

Hands-On Exercises

EXERCISE 1

Perform two writes through `write_page_with_wal()` to two different pages — let the first complete fully (logged and applied), then simulate a crash right after logging the second (never calling `heap_file.write_page()` for it). Verify the data file has page 1's update but not page 2's, while the WAL contains records for *both*. Explain why it has to be safe for a future recovery routine to reapply page 1's own record too, even though it was already applied.

View solution

EXERCISE 2

Write two records to a WAL file, close it, then simulate a crash by truncating the file so that only the first record survives (matching Finding 2's own torn-write setup). Open a brand-new `WAL` object pointed at the same file and confirm `next_lsn` correctly continues from the surviving first record's own LSN, not from any value that could be parsed out of the torn second record.

View solution

EXERCISE 3

Corrupt a single byte *inside* a record's own 4-byte length field (rather than inside its payload, as Finding 2b did) and confirm how `read_all_records()` responds. Explain, in your own words, which of the two checks — the bounds check or the checksum — actually catches this specific kind of corruption, and why.

View solution

CHAPTER 2 QUICK REFERENCE

- **A WAL record:** length-prefixed and checksummed — `[4B length][4B CRC32][LSN][page_num][new page bytes]`
- **The rule:** log first (durably, via `os.fsync()`), apply to the data file second — never the other way around

- **Verified Finding 1:** a "crashed" write leaves the data file honestly untouched, but the WAL's own record of it survives completely intact and byte-exact
- **Verified Finding 2:** a naive reader with no bounds check or checksum doesn't crash on a torn trailing record — it silently fabricates a plausible-looking, genuinely corrupted one
- **Verified Finding 2b:** a length/bounds check alone can't catch in-place corruption — only a checksum can
- **Next chapter:** Crash Recovery — actually replaying a WAL's own valid records to rebuild correct state after a real simulated crash

CHAPTER 3 OF 10

Crash Recovery: Replaying the Log

Building a Database Engine: Transactions & Concurrency

Chapter 3 · Crash Recovery: Replaying the Log

Chapter 2 built a log that survives a crash. It never actually used that log for anything — the whole point of Chapter 2's own Finding 1 was that the data file stayed wrong even though the WAL was perfectly intact. This chapter closes that loop: read every record the log can prove is real, and reapply it to the data file, on startup, before anything else runs.

The Real Recovery Routine

```
def recover(heap_file, wal_path):
    records = read_all_records(wal_path)      # Chapter 2's own checksum-verified reader
    records.sort(key=lambda r: r['lsn'])
    for r in records:
        page = load_page(r['new_page_bytes'])
        heap_file.write_page(r['page_num'], page)
```

That's the whole mechanism. It has no idea which of these records were already applied before the crash and which weren't — and, as the next finding shows, it doesn't need to.

Finding 1: A Crashed Write, Actually Restored

Chapter 2's own Finding 1 logged an update, simulated a crash before it was applied, and confirmed the data file stayed untouched while the WAL held a complete, correct record of what was meant to happen. Running `recover()` against that exact same scenario closes the gap.

VERIFIED DIRECTLY — THE UPDATE IS GENUINELY RESTORED

Before recovery: the target page's bytes are still blank, exactly as Chapter 2 left them. After calling `recover(heap_file, wal_path)`: the page matches the intended update *exactly*, and `recover()` reports **1** record applied. What Chapter 2 could only prove survived in the log now genuinely exists in the data file.

Finding 2: Replaying an Already-Applied Record Is Safe

A real crash rarely interrupts a single write in isolation — by the time it happens, some earlier writes in the log have usually already made it all the way to the data file. Chapter 2's own Exercise 1 built exactly this scenario: one page fully written and applied, a second page logged but never applied, both sitting in the same WAL.

VERIFIED DIRECTLY — REAPPLYING A CORRECT RECORD CHANGES NOTHING

Running `recover()` over a WAL containing both records applies **both** — including page 1's own record, even though page 1 was already correct before recovery ever ran. After recovery, both pages match their intended data exactly, with no different or wrong outcome from having replayed the already-correct one a second time. Recovery never has to work out which records are "new" — it can unconditionally replay everything, every time, because Chapter 2's own decision to log the complete new page rather than a delta makes reapplying an already-applied record a genuine no-op.

Finding 3: Truncating the Log Too Early Can Lose Data Forever

A WAL that's never cleared grows without bound — the obvious next step is to truncate it once recovery has replayed everything, reclaiming the space. The obvious place to put that truncation call is at the *start* of recovery, right after reading the records out: it feels safe, since the records are already sitting in a local Python list by then.

```
def recover_naive_truncate_first(heap_file, wal_path):
    records = read_all_records(wal_path)
    records.sort(key=lambda r: r['lsn'])
    open(wal_path, 'wb').close()          # BUG: clears the log before replay is confirmed done
    for r in records:
        page = load_page(r['new_page_bytes'])
        heap_file.write_page(r['page_num'], page)
```

This looks completely reasonable — the records are already read into memory, so what could truncating the file underneath them possibly break? The test: log three writes, apply none of them, then simulate a *second* crash during recovery itself — right after the first record has been applied, before the other two.

VERIFIED DIRECTLY — PAGES 1 AND 2'S OWN UPDATES ARE GENUINELY, PERMANENTLY GONE

After the second crash: page 0 (applied before the interruption) is correct. Pages 1 and 2 — never reached — are still blank. And `read_all_records(wal_path)` now returns **zero** records: the log was already truncated, at the very start of this same recovery run, before any of the three records had actually been confirmed applied. The updates for pages 1 and 2 exist nowhere at all anymore — not in the data file, not in the log. This is the exact durability failure Chapter 2's entire WAL was built to prevent, reintroduced by the recovery routine meant to fix it.

The Fix: Truncate Last, Only Once Everything Is Confirmed Durable

```
def recover_safe_truncate_last(heap_file, wal_path):
    records = read_all_records(wal_path)
    records.sort(key=lambda r: r['lsn'])
    for r in records:
        page = load_page(r['new_page_bytes'])
        heap_file.write_page(r['page_num'], page)
    heap_file.file.flush()
    os.fsync(heap_file.file.fileno())        # confirm every applied page is durably on disk
    open(wal_path, 'wb').close()          # ONLY NOW is it safe to clear the log
```

VERIFIED DIRECTLY — THE SAME INTERRUPTION, NO DATA LOST

Simulating the identical second crash (same interruption point, same three records) against this version: right after the crash, `read_all_records(wal_path)` still returns all **3** records — the truncate call is the very last line in the function, and was never reached. Simply re-running `recover_safe_truncate_last()` from scratch — no special "resume from where I left off" logic anywhere — applies all three records correctly (redoing page 0's own record harmlessly, per Finding 2), and only *then*, with everything genuinely durable, does the log finally get cleared.

Finding 2 and Finding 3 depend on each other directly: it's only *because* replaying an already-applied record is provably safe that "just keep the log around and re-run recovery in full after any interruption" is a correct strategy at all, rather than something that risks corrupting already-correct pages.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Recovery closing Chapter 2's own crash-survival gap	Chapter 2 (this course) — the exact scenario built there is resolved here, not re-explained from scratch
Idempotent replay of already-applied records	Chapter 2's own Exercise 1 — the "WAL ahead of the data file" scenario built there is directly reused as this chapter's own opening test
Truncating too early destroying otherwise-recoverable data	Building a Database Engine: Storage & Query Fundamentals Chapter 4 (Course 1) — the dormant, never-written page-header bug found there was also a case of code that looked obviously safe until a real restart exposed it

Hands-On Exercises

EXERCISE 1

Run a write through completely (logged, applied, heap file fsynced, WAL cleanly truncated), then call `recover()` against that now-empty WAL. Confirm it applies zero records and leaves the already-correct page completely unchanged.

View solution

EXERCISE 2

Log three separate updates to the *same* page (three genuinely different versions of its content), never apply any of them, then run `recover()`. Confirm all three replay in LSN order and the final page matches the *last* one logged — exactly as if the original three writes had never been interrupted at all.

View solution

EXERCISE 3

Log updates to two different pages, simulate a crash inside `recover()` itself right after the first page is applied but before the second. Confirm the WAL is untouched by this plain `recover()` (unlike the naive truncate-first version), then re-run `recover()` a second time and confirm both pages end up correct.

View solution

CHAPTER 3 QUICK REFERENCE

- **Recovery:** read every valid WAL record, sort by LSN, reapply each one to the heap file — unconditionally, every time
- **Verified Finding 1:** a write Chapter 2 showed surviving only in the log is now actually restored to the data file
- **Verified Finding 2:** reapplying an already-applied record is a genuine no-op — safe because each record logs the complete new page, not a delta
- **Verified Finding 3:** truncating the WAL before replay is confirmed complete can permanently lose data if recovery itself is interrupted — fix: fsync the data file, *then* truncate the log, never the other way around
- **Next chapter:** Atomicity — undo logging and rollback, closing the multi-step "insert a row, then update its indexes" gap Chapter 1 opened this course with

CHAPTER 4 OF 10

Atomicity: Undo Logging & Rollback

Building a Database Engine: Transactions & Concurrency

Chapter 4 · Atomicity: Undo Logging & Rollback

This course opened with a real, verified bug: a two-step insert — write the row, then update the index — that leaves the table and its own index silently disagreeing if anything interrupts it between the two steps. Chapters 2 and 3 closed the durability half of that story: a crash before a write is applied is now recoverable. But Chapter 1's own original scenario wasn't a crash — it was one step succeeding and the next one failing outright. This chapter closes that half: wrap the whole operation in a transaction, and if any part of it fails, undo everything that already happened.

A Real Undo Log

Where Chapter 2's WAL recorded each page's *new* content, an undo log records each page's *old* content — captured the first time, and only the first time, a transaction touches it:

```
class Transaction:
    def __init__(self):
        self.undo_log = []      # [(file_name, page_num, old_bytes_or_None), ...]
        self.touched = set()    # {(file_name, page_num)} -- captured ONCE per page

    def record_write(self, file_name, real_heap_file, page_num):
        key = (file_name, page_num)
        if key in self.touched:
            return                # already captured -- this write doesn't need a new record
        if page_num < real_heap_file.num_pages():
            old_bytes = bytes(real_heap_file.read_page(page_num).data)
        else:
            old_bytes = None       # this page didn't exist before the transaction touched it
        self.undo_log.append((file_name, page_num, old_bytes))
        self.touched.add(key)

    def commit(self):
        self.undo_log = []; self.touched = set()      # discard -- the changes stand

    def rollback(self, real_heap_files):
        for file_name, page_num, old_bytes in reversed(self.undo_log):
            heap_file = real_heap_files[file_name]
            if old_bytes is not None:
                heap_file.write_page(page_num, load_page(old_bytes))
            else:
                heap_file.write_page(page_num, Page())    # blank it -- see below
        self.undo_log = []; self.touched = set()
```

Making the write side of every table transparently participate needs no changes at all to Course 1's own `HeapTable` — only a thin wrapper implementing the exact same interface, so `HeapTable` genuinely has no idea it's being run inside a transaction:

```
class TransactionalHeapFile:
    def __init__(self, real_heap_file, transaction, file_name):
        self.real = real_heap_file
        self.transaction = transaction
        self.file_name = file_name
    def num_pages(self): return self.real.num_pages()
    def read_page(self, page_num): return self.real.read_page(page_num)
    def write_page(self, page_num, page):
        self.transaction.record_write(self.file_name, self.real, page_num)
        self.real.write_page(page_num, page)
    def allocate_page(self):
        page_num = self.num_pages()
        self.write_page(page_num, Page())   # goes through OUR write_page -- captured too
        return page_num
```

Finding 1: Commit Keeps Changes, Rollback Fully Undoes Them

VERIFIED DIRECTLY — TWO INDEPENDENT PAGES, ROLLED BACK CLEANLY

A transaction writes new content to two different pages, then `rollback()` is called instead of `commit()`. Both pages come back byte-for-byte identical to their state before the transaction began — the writes are genuinely undone, not merely left inconsistent.

Finding 2: First-Touch Capture Restores the TRUE Original, Not an Intermediate State

A single transaction writes to the *same* page twice — once with content `V1`, then again with different content `V2`.

VERIFIED DIRECTLY — ONE UNDO RECORD, NOT TWO

The undo log ends up with exactly **1** entry for that page, not 2 — the second write finds the page already in `self.touched` and skips capturing anything. After `rollback()`, the page matches its true pre-transaction state exactly, and is confirmed *not* equal to `V1` — the intermediate state the page passed through partway through the transaction. Capturing on first touch only means rollback always reaches all the way back to where the transaction began, no matter how many times a page was rewritten along the way.

Finding 3: The Payoff — Genuinely Resolving Chapter 1's Own Bug

Chapter 1's own opening scenario, unmodified: insert a row, then update an index for it. This time, both steps run inside a transaction, and the index update is made to fail on purpose.

```
def transactional_insert_with_index(txn, real_heap_file, schema, values, index_root, should_fail):
    thf = TransactionalHeapFile(real_heap_file, txn, 'accounts')
    table = HeapTable(thf, schema)              # Course 1's own HeapTable, completely unmodified
    location = table.insert(values)             # STEP 1 -- goes through the transactional wrapper
    if should_fail:
        raise RuntimeError("simulated failure updating the index")
    new_root = btree_insert(index_root, values[0], location)   # STEP 2
    return location, new_root

txn = Transaction()
try:
    transactional_insert_with_index(txn, real_heap, schema, [2, 'Bob', 300], index_root, should_fail=
except RuntimeError:
    txn.rollback({'accounts': real_heap})
```

VERIFIED DIRECTLY — THE ROW WAS NEVER INSERTED, AS FAR AS THE TABLE CAN TELL

After catching the exception and calling `rollback()`, a real `scan()` of the table finds **zero** rows. Compare this against Chapter 1's own Finding 1, where the exact same interrupted-operation shape left a row genuinely, permanently present but unindexed. Here, the row insertion itself is undone along with everything else — the table is back to exactly where it was before the attempt, with no inconsistency at all. Running the same operation with `should_fail=False` and calling `commit()` instead confirms the successful path still works correctly too: the row is present, and the index correctly points at its real location.

Finding 4: Why a Non-Deduped Undo Log Must Replay in Reverse

This chapter's own `Transaction` class sidesteps a subtle ordering problem entirely, by construction, since it never keeps more than one undo record per page. A log that captures *every* write — not just the first one per page — doesn't get that guarantee for free.

VERIFIED DIRECTLY — FORWARD-ORDER REPLAY LEAVES THE PAGE AT AN INTERMEDIATE STATE

Two writes to the same page (`V1`, then `V2`) are logged without deduplication — two separate undo entries: `(before V1)` and `(before V2, i.e. V1)`. Replaying them in the order they were *captured* — the natural-looking first instinct — applies `(before V1)` first, then `(before V2)`, landing the page on `V1`, not the true original. Replaying the exact same two records in **reverse** order lands correctly on the true pre-transaction state. "Undo" means walking history backward — a log that isn't already deduplicated to one entry per page depends on that being done explicitly, not assumed.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Genuinely atomic insert-plus-index-update	Chapter 1 (this course) — the exact scenario opened there is fully resolved here, closing the loop this entire course started with
Reverse-order replay for a non-deduped log	Chapter 3's own `recover()` — a structurally similar "replay records in the right order" concern, though redo (Chapter 3) and undo (this chapter) need *opposite* orderings for the same underlying reason
A newly-allocated page that can't be shrunk back on rollback	An honest, deliberate scope limitation of this engine — real systems solve this with a free-page/deallocation map, not attempted here

Hands-On Exercises

EXERCISE 1

Within a transaction, allocate a brand-new page (via `TransactionalHeapFile.allocate_page()`) and write a row to it, then call `rollback()`. Confirm `num_pages()` does *not* shrink back down (the file itself can't reclaim the space), but the page's own content is blanked and a real `scan()` finds no rows there.

View solution

EXERCISE 2

Within a transaction, *read* a page (via `TransactionalHeapFile.read_page()`) but never write to it, then write to a *different* page and roll back. Confirm the undo log contains exactly one entry — for the written page only — and that the read-only page's own real content is completely unaffected by the rollback.

View solution

EXERCISE 3

Write to the same page *three* times within one transaction (three genuinely different contents), then roll back. Confirm the undo log still contains only one entry (not three), and that the page ends up matching its true pre-transaction state, not any of the three intermediate versions.

View solution

CHAPTER 4 QUICK REFERENCE

- **Undo log:** captures each page's before-image, but only on its *first* touch within a transaction
- **rollback():** restores every touched page to its pre-transaction state, in reverse order
- **Verified Finding 1:** rollback fully undoes writes across multiple pages
- **Verified Finding 2:** first-touch capture restores the TRUE original, not an intermediate state, no matter how many times a page was rewritten
- **Verified Finding 3 — the payoff:** wrapping Chapter 1's own insert-plus-index-update in a transaction and rolling back on failure leaves the table genuinely, completely unchanged — no more silent index/data inconsistency
- **Verified Finding 4:** a non-deduped undo log MUST replay in reverse order, or it lands on an intermediate state instead of the true original
- **Next chapter:** Isolation, Part 1 — locking, and a real, verified deadlock

CHAPTER 5 OF 10

Isolation, Part 1: Locking

Building a Database Engine: Transactions & Concurrency

Chapter 5 · Isolation, Part 1: Locking

Every chapter so far has run one operation at a time, alone, then checked what a crash or a failure would have done to it. Isolation is a genuinely different question: what happens when two operations run *at the same time*, for real? This chapter uses real Python threads — not a simulation of concurrency, actual concurrent execution — to reproduce a genuine race condition, fix it with real locks, and then reproduce a genuine deadlock.

A Real Lock Manager: Shared and Exclusive

```
class LockManager:
    def __init__(self):
        self.cond = threading.Condition(threading.Lock())
        self.exclusive_holder = {}   # page_key -> txn_id
        self.shared_holders = {}     # page_key -> set(txn_id)

    def acquire_shared(self, page_key, txn_id):
        with self.cond:
            while True:
                holder = self.exclusive_holder.get(page_key)
                if holder is None or holder == txn_id:
                    self.shared_holders.setdefault(page_key, set()).add(txn_id)
                    return
                self.cond.wait()      # someone else holds it EXCLUSIVELY -- wait

    def acquire_exclusive(self, page_key, txn_id):
        with self.cond:
            while True:
                holder = self.exclusive_holder.get(page_key)
                sharers = self.shared_holders.get(page_key, set()) - {txn_id}
                if (holder is None or holder == txn_id) and not sharers:
                    self.exclusive_holder[page_key] = txn_id
                    return
                self.cond.wait()      # someone else holds it, shared or exclusive -- wait
```

Shared locks (for reads) can be held by multiple transactions on the same page at once. An exclusive lock (for writes) needs the page completely to itself — no other shared or exclusive holder at all.

Finding 1: A Real, Reproducible Lost Update

Five real threads, each incrementing one shared counter twenty times — read the current value, sleep briefly (to force a genuine interleaving window), write back `value + 1`. No locking at all.

VERIFIED DIRECTLY — REAL INCREMENTS GENUINELY GO MISSING

Expected final value: **100** (5 threads × 20 increments). Actual measured result: as low as **20**, varying run to run — always short of 100. This isn't a theoretical risk described in the abstract; it's a real, measured shortfall. Two threads reading the same value before either has written its own increment back means one thread's update is silently overwritten by the other's — a genuine lost update, reproduced on demand.

Finding 2: The Same Test, Correct Every Time With a Real Lock

VERIFIED DIRECTLY — AN EXCLUSIVE LOCK HELD ACROSS THE WHOLE READ-MODIFY-WRITE CYCLE

The identical test, with `lock_mgr.acquire_exclusive('counter', txn_id)` held from just before the read to just after the write, and released only then: the final value is **100** — exactly correct, every single run. Holding the lock across the *entire* critical section, not just the write itself, is what closes the gap Finding 1 exploited.

Finding 3: Shared Locks Overlap, Exclusive Locks Serialize

Four threads each hold a lock on the same page for a fixed `0.15`-second duration — once with a shared lock, once with an exclusive lock — measuring real wall-clock time for all four to finish.

VERIFIED DIRECTLY — REAL TIMING CONFIRMS THE TWO LOCK MODES BEHAVE COMPLETELY DIFFERENTLY

Four threads holding a **shared** lock: real elapsed time ≈ `0.151s` — essentially *one* hold-duration, since all four genuinely overlap. Four threads holding an **exclusive** lock: real elapsed time ≈ `0.602s` — close to *four* hold-durations, since each one has to wait for the previous one to finish. The two lock modes aren't just conceptually different — they produce measurably different real-world timing.

Finding 4: A Real, Reproducible Deadlock

Two transactions, each locking one page first, then trying to lock the *other's* page — the classic circular-wait shape. Transaction A locks page 1, then tries for page 2. Transaction B locks page 2, then tries for page 1.

VERIFIED DIRECTLY — BOTH SIDES GENUINELY STUCK, CONFIRMED VIA A BOUNDED TIMEOUT

Both transactions' second lock attempt uses a 1-second timeout (since a real test can't wait forever to prove something never finishes). Both attempts report **failure** — neither one acquires its second lock — and the real elapsed time is `≈1.0s`, matching the timeout exactly rather than resolving early. Each transaction is waiting for a lock the other one holds, and neither will release what it's already holding until it gets what it's waiting for. Without a timeout, this genuinely never resolves on its own.

Finding 5: Wait-for-Graph Detection Catches the Cycle Immediately

A deadlock detector doesn't need to wait for anything to time out — it needs to notice, at the moment a transaction is *about* to wait, that doing so would close a cycle: "I'm about to wait for you, and you're already waiting for me."

```
def _creates_cycle(self, waiter, blockers):
    visited = set()
    stack = list(blockers)
    while stack:
        node = stack.pop()
        if node == waiter:
            return True          # found our way back to ourselves -- a cycle
        if node in visited:
            continue
        visited.add(node)
        stack.extend(self.waits_for.get(node, set()))
    return False
```

VERIFIED DIRECTLY — ONE SIDE ABORTED INSTANTLY, THE OTHER PROCEEDS WITHOUT EVER WAITING ON A TIMEOUT

The same two-transaction setup as Finding 4, but each acquire call now checks `_creates_cycle()` before blocking. Transaction A gets there first, finds no cycle, and legitimately starts waiting for page 2. Transaction B's own attempt for page 1 is checked next: the wait-for graph shows A is already waiting on B, so B waiting on A would close the loop — detected immediately, and B's call **raises** rather than blocking. B releases the one lock it does hold (the same real action `rollback()` from Chapter 4 would take for an aborted transaction), which wakes A up, and A's own attempt succeeds. Real elapsed time: `≈0.001s` — nowhere near the 1-second timeout Finding 4 needed to prove its own point.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
The lost-update race condition	Chapter 1 (this course) — the third named, still-untested ACID gap ("two operations running at the same time") is finally exercised for real, with real threads, in this chapter
Aborting the deadlock's losing side by releasing its locks	Chapter 4 (this course) — a real, concrete instance of exactly what `rollback()` is for: undoing a transaction's own work when it can't be allowed to continue
Real threading used to reproduce a genuine race condition	Design Patterns (this site's Software Development subject) — the same technique (real threads, a deliberate sleep to force interleaving) reproduced a genuine Singleton race condition there

Hands-On Exercises

EXERCISE 1

Have one thread acquire a shared lock on a page and hold it for 0.3 seconds, while a second thread waits for the first to actually have the lock and then tries to acquire an *exclusive* lock on that same page. Measure how long the second thread's own acquire call takes, and confirm it genuinely blocks for close to the full 0.3 seconds rather than succeeding immediately.

View solution

EXERCISE 2

Reproduce a deadlock among *three* transactions instead of two — A waits on B, B waits on C, C waits on A, closing a longer cycle. Use a real `threading.Barrier` to guarantee all three first locks are granted before any second attempt begins, and confirm all three attempts genuinely time out.

View solution

EXERCISE 3

Have two transactions each need locks on the same two pages, but have both transactions acquire them in a single, *consistent* global order (e.g., always sorted by page key) rather than in whatever order each transaction happens to want them. Confirm both transactions succeed quickly, with no deadlock and no detector needed at all.

View solution

CHAPTER 5 QUICK REFERENCE

- **Shared lock:** many transactions can hold it on the same page at once (reads)

- **Exclusive lock:** only one transaction can hold it, and no one else can hold any lock at all on that page (writes)
- **Verified Finding 1:** a real, reproducible lost update — concurrent read-modify-write with no locking genuinely loses increments
- **Verified Finding 2:** the same test is correct every time once an exclusive lock covers the whole critical section
- **Verified Finding 3:** shared locks overlap in real elapsed time; exclusive locks serialize
- **Verified Finding 4:** two transactions locking pages in opposite orders genuinely deadlock — confirmed by both sides staying blocked past a bounded timeout
- **Verified Finding 5:** a wait-for-graph detector catches the cycle immediately and aborts one side, rather than waiting on any timeout at all
- **Next chapter:** Isolation, Part 2 — MVCC, a real alternative to locking altogether

CHAPTER 6 OF 10

Isolation, Part 2: An Introduction to MVCC

Building a Database Engine: Transactions & Concurrency

Chapter 6 · Isolation, Part 2: An Introduction to MVCC

Chapter 5 solved isolation with locking: a reader and a writer touching the same page genuinely have to wait for each other. MVCC — multi-version concurrency control — solves the same problem a completely different way: instead of one mutable value per row that everyone fights over, keep every version a row has ever had, tagged with when it was created and when it stopped being current. A reader just picks the version that was current as of the moment its own transaction began, and never has to wait for anyone.

A Real Version Chain, Not One Mutable Value

```
class Version:
    def __init__(self, value, created_by):
        self.value = value
        self.created_by = created_by   # txn_id that wrote this version
        self.created_ts = None         # commit sequence number -- None until committed
        self.deleted_ts = None         # commit sequence number when superseded -- None if still curr

class MVCCStore:
    def __init__(self, detect_conflicts=False):
        self._next_ts = 1
        self.versions = {}             # row_id -> [Version, ...] in commit order
        self.txn_snapshot = {}         # txn_id -> snapshot ts, captured at begin()
        self.txn_pending_writes = {}   # txn_id -> {row_id: Version} -- uncommitted
```

Reading a row means finding the newest version whose `created_ts` is at or before the reader's own snapshot, and which hadn't yet been superseded (`deleted_ts`) as of that same snapshot. Writing never touches an existing version at all — it stages a brand-new one, invisible to everyone else until commit.

A Real Bug Found Building the Very First Test: the Snapshot Boundary

The first, most natural way to compute a transaction's own snapshot at `begin()` looks completely reasonable: `self.txn_snapshot[txn_id] = self._next_ts` — "whatever the counter currently reads." It's

wrong.

VERIFIED DIRECTLY — A READER'S OWN SNAPSHOT APPEARED TO MOVE ON ITS OWN

A reader begins, reads a row (sees `100`), a completely separate transaction commits a new value (`999`), and the reader reads the same row again — expecting to still see `100`, since its own transaction never touched anything else. It saw `999` instead. The bug: `_next_ts` is the counter value the *next* commit will claim, not the value of the *last* commit that actually happened. A reader beginning right after a commit, and a writer whose commit lands moments later, could end up sharing the exact same timestamp — making it genuinely ambiguous whether the writer's commit happened "before" or "after" the reader's own snapshot.

THE FIX — SNAPSHOT = THE LAST TS ACTUALLY COMMITTED, NOT THE NEXT ONE AVAILABLE

`self.txn_snapshot[txn_id] = self._next_ts - 1`. A transaction's snapshot should mean "everything committed strictly before I began," never "everything up to and including some commit that might land a moment from now." With this fix, the exact same test correctly returns `100` both times — the reader's snapshot genuinely never moves, no matter what commits happen around it.

Finding 1: A Stable Snapshot, Verified Against a Real Concurrent Commit

VERIFIED DIRECTLY — THE FIXED VERSION BEHAVES CORRECTLY

Reader begins, reads `100`. A separate transaction writes and commits `999`. Reader reads again: still `100`. A **brand-new** transaction started after that commit reads `999` immediately. The old reader isn't blocking the writer, and isn't blocked by it either — each transaction simply sees the world as of its own snapshot.

Finding 2: Read-Your-Own-Writes

VERIFIED DIRECTLY — VISIBLE TO YOURSELF IMMEDIATELY, INVISIBLE TO EVERYONE ELSE UNTIL COMMIT

A transaction writes `42` to a row and reads it back before committing: sees `42`. A different transaction reading the same row at the same moment: sees the old, still-committed value, `1`. Uncommitted work is real to the transaction that did it, and doesn't exist to anyone else yet.

Finding 3: Real Threads — Neither Side Ever Waits

A long-running reader holds a transaction open for `0.15`s, reading the same row twice. A separate thread commits a new value to that row partway through.

VERIFIED DIRECTLY — REAL, MEASURED TIMING CONFIRMS NEITHER SIDE BLOCKS

The reader's two reads both return the original value, `'A'` — its snapshot never moves. The writer's own `commit()` call completes in **0.01ms** — it never had to wait for anything. Total wall time for both threads together: **≈0.15s**, roughly *one* hold-duration, not two. Compare this directly against Chapter 5's own Finding 3, where an exclusive writer genuinely had to wait out a shared lock's own full hold-time before it could even begin.

Finding 4: An Honest Limitation — Version History Grows Forever

VERIFIED DIRECTLY — NOTHING IS EVER CLEANED UP

Fifty separate commits to the same row, one after another: `len(store.versions['counter'])` reports **51** — every single version, including the 50 that no transaction beginning from this point forward could possibly still need. This engine has no garbage collection for old versions at all. Real systems need one — PostgreSQL's own `VACUUM` process exists specifically to remove versions no active transaction's snapshot could ever reach anymore. Not attempted here; a real, deliberate scope gap.

Finding 5: MVCC Alone Doesn't Prevent a Lost Update

Two transactions both begin from the same snapshot, both read `100`, one computes `+10` and the other computes `+20`, and both commit.

VERIFIED DIRECTLY — CHAPTER 5'S OWN RACE CONDITION, REPRODUCED WITH ZERO LOCKING ANYWHERE

Both commits report success. The final balance is `120`, not `130` — the first transaction's own `+10` is silently gone, overwritten by the second commit, which never knew the first one had happened. Nothing anywhere raised an error. MVCC solved concurrent *reads* never blocking — it never claimed to solve concurrent *writes* to the same row on its own.

```
def commit(self, txn_id):
    snapshot = self.txn_snapshot[txn_id]
    if self.detect_conflicts:
        for row_id in self.txn_pending_writes[txn_id]:
            current = self._current_committed_version(row_id)
            if current is not None and current.created_ts > snapshot:
                raise WriteConflict(
                    f"row {row_id!r} was committed by another transaction after this txn's own snapsh
                )
    # ... proceed with the commit as before ...
```

VERIFIED DIRECTLY — FIRST-COMMITTER-WINS CATCHES IT

With the conflict check active, the first transaction commits normally (`100 → 110`). The second transaction's own commit is **rejected** — the row it read from was already superseded by a commit that happened after its own snapshot. It must retry: begin a fresh transaction, re-read the real current value (`110`), and redo its own calculation from there.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Never blocking, verified with real threads	Chapter 5's own Finding 3 — the identical timing-based verification technique, this time proving the opposite property (no wait, not "correctly serialized")
A lost update reproduced under MVCC	Chapter 5's own Finding 1 — the same underlying race, appearing under a completely different concurrency strategy, resolved with a conflict check rather than a lock
Unbounded version growth	An honest, deliberate scope gap — a real vacuum/garbage-collection pass is a substantial feature of its own, not attempted in this course

Hands-On Exercises

EXERCISE 1

With conflict detection enabled, run a transaction that only *reads* a row (never writes to it), let a separate transaction commit a change to that same row, and then commit the read-only transaction. Confirm it commits with no conflict at all, and explain why a read-only transaction can never trigger the conflict check.

View solution

EXERCISE 2

With conflict detection enabled, run two transactions from the same starting snapshot, but have each one write to a *different* row instead of the same one. Confirm both commit successfully with no conflict, and explain why the conflict check being per-row (not per-transaction or per-database) makes this the correct outcome.

View solution

EXERCISE 3

Reproduce Finding 5's own lost-update scenario with conflict detection enabled, catch the resulting `WriteConflict` , and then *retry* the losing transaction properly — begin a brand-new transaction, re-read the current value, and redo the calculation. Confirm the final balance correctly reflects both increments this time, with nothing lost.

View solution

CHAPTER 6 QUICK REFERENCE

- **MVCC's core idea:** keep every version of a row, tagged with when it was created and superseded — readers pick the version current as of their own snapshot
- **Real bug found:** a transaction's snapshot must be the last commit ts that already happened, not the next one available — off by one, and a genuinely wrong snapshot boundary
- **Verified Finding 1:** a reader's snapshot stays stable across concurrent commits; a fresh transaction sees the new value immediately
- **Verified Finding 2:** read-your-own-writes — visible to yourself before commit, invisible to everyone else
- **Verified Finding 3:** real threads confirm neither a long reader nor a concurrent writer ever blocks the other
- **Verified Finding 4:** version history grows without bound — no garbage collection, an honest scope gap
- **Verified Finding 5:** MVCC alone still allows a lost update between concurrent writers to the same row — fixed with a first-committer-wins conflict check
- **Next chapter:** Multi-Table Support & Foreign Keys — extending this engine beyond a single table

CHAPTER 7 OF 10

Multi-Table Support & Foreign Keys

Building a Database Engine: Transactions & Concurrency

Chapter 7 · Multi-Table Support & Foreign Keys

Course 1's own `Database` class could already hold more than one table — that was never the missing piece. What's missing is any guarantee about how those tables relate to each other: nothing stops an `orders` row from pointing at a `customer_id` that doesn't exist anywhere in `customers`. This chapter builds a real foreign key constraint — checked on every insert, using the exact same B-tree index machinery Course 1 already built for a completely different reason.

A Real Catalog, and a Real Constraint

```
class ForeignKey:
    def __init__(self, column, ref_table, ref_column):
        self.column = column; self.ref_table = ref_table; self.ref_column = ref_column

class Catalog:
    def insert(self, table_name, values):
        schema, table = self.tables[table_name]
        column_names = [n for n, t in schema.columns]
        for fk in self.foreign_keys.get(table_name, []):
            idx = column_names.index(fk.column)
            fk_value = values[idx]
            if not self._value_exists(fk.ref_table, fk.ref_column, fk_value):
                raise ForeignKeyViolation(
                    f"{fk.column}={fk_value!r} does not exist in {fk.ref_table}.{fk.ref_column}"
                )
        location = table.insert(values)     # only reached if every FK check passed
        # ... update indexes for this table, unchanged from Course 1 Chapter 9 ...
        return location
```

Finding 1: A Real Foreign Key, Verified Both Ways

VERIFIED DIRECTLY — A VALID INSERT SUCCEEDS, AN INVALID ONE IS REJECTED AND NEVER TOUCHES DISK

An order referencing customer `id=1` (a real, existing customer) inserts normally. An order referencing customer `id=999` (which doesn't exist) raises `ForeignKeyViolation` — and a real scan of the orders table afterward confirms it: only the valid order is present. The rejected insert never reached `table.insert()` at all.

Finding 2: Reusing Course 1's Own Index Makes This Fast

Checking whether a value exists in another table means either scanning it in full, or — if an index already exists on the referenced column — a real `btree_search()` call.

VERIFIED DIRECTLY — A REAL, MEASURED 180× SPEEDUP

200 foreign-key-checked inserts against a 3,000-row `customers` table, always referencing the last row (the worst case for a scan): **0.617s** with no index. The identical test with an index on `customers.id`: **0.003s** — **≈180×** faster. A foreign key check is nothing more than a repeated existence lookup — exactly the operation Course 1's own Storage & Query Fundamentals Chapters 5–6 built a B-tree index to make fast.

Finding 3: A Real Bug — Referencing a Table That Doesn't Exist Yet

The first, most natural version of `create_table()` just stores whatever foreign keys it's given, with no check that the referenced table actually exists.

VERIFIED DIRECTLY — A CONFUSING, UNHELPFUL CRASH ON THE FIRST INSERT

Creating `orders` with a foreign key to `customers` — *before* `customers` has ever been created — succeeds with no complaint at all. The real failure only shows up later, on the very first insert into `orders`, as a raw `KeyError: 'customers'` — a confusing error with no indication of what actually went wrong or when the real mistake happened.

THE FIX — VALIDATE THE REFERENCE IMMEDIATELY, AT CREATE TABLE TIME

`create_table()` now checks every foreign key's own `ref_table` against the catalog's existing tables (allowing `ref_table == name` for a genuine self-reference) before registering anything. The identical mistake now raises a clear `SchemaError` — "*foreign key references unknown table 'customers' — it must be created first*" — at the exact moment it's made, and `orders` is never left half-registered in the catalog.

Finding 4: A Real, Working Multi-Table System

VERIFIED DIRECTLY — THREE CUSTOMERS, THREE VALID ORDERS, THREE REAL REJECTIONS

Three real customers, three valid orders referencing them, and three deliberately invalid `customer_id` values — all three rejected, none reaching the heap file. A manual join (matching each order's own `customer_id` against the

customers table) never has to handle a dangling reference, because the foreign key constraint guarantees none can exist.

A Genuine Structural Discovery: Self-Reference Under Check-Before-Insert

Building an exercise around a self-referencing table (an `employees` row whose own `manager_id` points at another row in the *same* table) surfaced something worth explaining honestly rather than working around quietly.

VERIFIED DIRECTLY — A ROW CAN NEVER REFERENCE ITS OWN NOT-YET-INSERTED ID

Inserting a row whose `manager_id` equals its *own* not-yet-assigned `id` — for instance, a CEO row that reports to itself — is **always rejected**, no matter which row number it is. This isn't a bug specific to self-reference: the foreign key check always runs *before* `table.insert()`, so the row genuinely doesn't exist yet at the moment its own value is checked against itself. Any row whose foreign key equals its own id is structurally impossible under this ordering.

Reordering to insert first and validate second — physically writing the row, then checking, then **rolling it back** if invalid — resolves this cleanly, reusing Chapter 4's own real `Transaction` and `TransactionalHeapFile` completely unmodified:

```
def insert_allow_self_reference(cat, table_name, values):
    schema, real_table = cat.tables[table_name]
    txn = Transaction()                                    # Chapter 4, unmodified
    thf = TransactionalHeapFile(real_table.heap_file, txn, table_name)
    location = HeapTable(thf, schema).insert(values)       # physically written first

    for fk in cat.foreign_keys.get(table_name, []):
        fk_value = values[[n for n, t in schema.columns].index(fk.column)]
        if not cat._value_exists(fk.ref_table, fk.ref_column, fk_value):
            txn.rollback({table_name: real_table.heap_file})     # undo the physical write
            raise ForeignKeyViolation(...)
    txn.commit()
    return location
```

VERIFIED DIRECTLY — SELF-REFERENCE NOW GENUINELY WORKS, AND TRUE VIOLATIONS STILL DON'T

A CEO row with `manager_id` pointing at its own `id` now inserts successfully — by the time the check runs, the row is already physically present, so it can find itself. A genuinely invalid row (referencing a manager who doesn't exist at all) is still correctly rejected, and Chapter 4's own `rollback()` cleanly removes the physical write it made moments earlier — the invalid row never appears in a final scan.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Index-backed FK checks, 180× faster	Building a Database Engine: Storage & Query Fundamentals Chapters 5–6, 9 (Course 1) — the B-tree index and query planner reused here for a genuinely new purpose
Insert-then-validate-then-rollback for self-reference	Chapter 4 (this course) — real, unmodified reuse of `Transaction` / `TransactionalHeapFile`, in a context that chapter never anticipated
Referencing a not-yet-created table	An honest, deliberate ordering constraint this engine now enforces — parent tables must exist before a child table can reference them, matching real SQL practice

Hands-On Exercises

EXERCISE 1

Create a table with no `foreign_keys` argument at all, insert a few rows into it, and confirm both that the inserts succeed exactly as they would in Course 1's own plain `HeapTable`, and that `catalog.foreign_keys[table_name]` is an empty list rather than `None` or missing entirely.

View solution

EXERCISE 2

Build the self-referencing `employees` example from this chapter yourself: confirm a row referencing its own not-yet-inserted id is rejected under the normal `insert()` path, then use `insert_allow_self_reference()` to successfully insert it, followed by a second, genuinely invalid row that should be rolled back.

View solution

EXERCISE 3

Build an index on a referenced column *before* any data exists in that table (on a genuinely empty table), then insert real rows into both tables afterward. Confirm the index stays accurate as rows are added, and that a foreign-key check against it still correctly catches an invalid reference.

View solution

CHAPTER 7 QUICK REFERENCE

- **ForeignKey(column, ref_table, ref_column):** checked on every insert, before the row is written

- **Verified Finding 1:** a valid insert succeeds; an invalid one is rejected and never reaches the heap file
- **Verified Finding 2:** routing the check through an existing B-tree index is ≈180× faster than a full scan
- **Verified Finding 3:** a foreign key to a not-yet-created table crashed confusingly on first insert — fixed with immediate validation at CREATE TABLE time
- **Genuine discovery:** a row can never reference its own not-yet-inserted id under check-before-insert — resolved by inserting first and rolling back on failure, reusing Chapter 4's own real transaction machinery
- **Next chapter:** Joins — nested-loop and hash join, replacing this chapter's manual dictionary-based join with real join algorithms

CHAPTER 8 OF 10

Joins: Nested-Loop & Hash Join

Building a Database Engine: Transactions & Concurrency

Chapter 8 · Joins: Nested-Loop & Hash Join

Chapter 7's own capstone joined customers to orders by hand — build a Python dict once, look each order up in it. That was already a hash join, just not named as one. This chapter builds two real join algorithms properly, verifies they agree with each other exactly, measures a real speed difference at scale, and finds a genuine correctness bug hiding in the "obvious" way to build a hash table.

Two Real Join Algorithms

```
def nested_loop_join(left_rows, left_key_idx, right_rows, right_key_idx):
    results = []
    for l in left_rows:
        for r in right_rows:          # a full inner scan for EVERY outer row
            if l[left_key_idx] == r[right_key_idx]:
                results.append((l, r))
    return results

def hash_join(build_rows, build_key_idx, probe_rows, probe_key_idx):
    hash_table = {}
    for row in build_rows:
        hash_table.setdefault(row[build_key_idx], []).append(row)   # a LIST per key
    results = []
    for probe_row in probe_rows:
        key = probe_row[probe_key_idx]
        for build_row in hash_table.get(key, []):
            results.append((build_row, probe_row))
    return results
```

Finding 1: The Two Algorithms Agree Exactly

VERIFIED DIRECTLY — IDENTICAL RESULT SETS, REAL DATA

Three real customers, four real orders (Alice has two, Bob and Carol have one each), joined via `nested_loop_join` and `hash_join`: both find exactly **4** pairs, and as sets they're identical. Two structurally very different algorithms produce

the same answer, as they must.

Finding 2: A Real, Measured 44× Speedup

VERIFIED DIRECTLY — REAL TIMING AT 800 × 800 ROWS

Nested-loop join: **0.0147s**. Hash join: **0.0003s** — **≈44×** faster, with identical result sets confirmed by set comparison. Nested-loop does a full 800-row inner scan for every one of 800 outer rows — `≈640,000` comparisons. Hash join builds one dict in 800 steps, then does 800 O(1) lookups — `≈1,600` steps total.

Finding 3: A Real, Silent Correctness Bug — Building on the Wrong Side

`customers.id` is unique — one row per id. `orders.customer_id` is *not* — Alice alone has two orders sharing the same key. The most natural first version of a hash join's own build step just assigns a key to a row, without thinking about what happens if that key shows up twice:

```
def hash_join_naive_overwrite(build_rows, build_key_idx, probe_rows, probe_key_idx):
    hash_table = {}
    for row in build_rows:
        hash_table[row[build_key_idx]] = row      # BUG: overwrites on a key collision
    # ...
```

VERIFIED DIRECTLY — ONE OF ALICE'S TWO REAL ORDERS SILENTLY VANISHES

Building the hash table on `orders` (the duplicate-key side), keyed by `customer_id`: the naive version finds only **3** pairs, not 4. Order `100` ($50, one of Alice's own two real orders) is completely missing — silently overwritten in the dict by order `101`, the last one inserted for `customer_id=1`. No error, no warning — just one fewer row in the final result than the true join actually has. Extending this with a customer who has *three* orders (Exercise 3) confirms the pattern generalizes: every duplicate past the first is silently dropped, not just one.

The fix is the same one shown in this chapter's own `hash_join` from the start: `hash_table.setdefault(key, []).append(row)` instead of a bare assignment. Every key maps to a *list* of rows, so a collision means "append," never "overwrite."

Finding 4: Once Fixed, Either Side Is Safe to Build On

VERIFIED DIRECTLY — CORRECTNESS NO LONGER DEPENDS ON WHICH SIDE YOU PICK

Building the correct, list-based hash table on `customers` (the unique-key side) and on `orders` (the duplicate-key side) both produce the exact same **4**-pair result, confirmed by set comparison. The real-world advice to build a hash join on the *smaller* table is about performance and memory — not correctness. Correctness only depends on never assuming a key is unique on the build side unless it genuinely, provably is.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Nested-loop's O(n×m) vs. hash join's O(n+m)	Maths for Programmers: Algorithms & Complexity — the exact same growth-rate reasoning, this time measured on a real join instead of an abstract example
A real customer, real orders, joined by real code	Chapter 7 (this course) — the manual dictionary-based join in that chapter's own capstone was already an unnamed hash join; this chapter names it, generalizes it, and finds the bug hiding in the shortcut version
"Confidently wrong is worse than a crash"	This site's own recurring theme — the naive hash join doesn't error, doesn't warn, just quietly returns fewer rows than the true answer

Hands-On Exercises

EXERCISE 1

Join a table of one customer against a table of one order whose `customer_id` doesn't match that customer's own id at all. Confirm both `nested_loop_join` and `hash_join` correctly return an empty result, with no error and no special-case handling needed in either.

View solution

EXERCISE 2

Instrument both join functions to count real operations — every comparison for nested-loop, every build-plus-probe step for hash join — and run them against this chapter's own 3-customer, 4-order data. Confirm the counts match `rows_left × rows_right` and `rows_left + rows_right` exactly, not just approximately.

View solution

EXERCISE 3

Give one customer *three* orders instead of two, and run the naive overwrite hash join built on the orders side. Confirm it now finds only 1 pair instead of the true 3, and explain why the number of silently dropped rows scales with the number of duplicates for that key, not capped at just one.

View solution

CHAPTER 8 QUICK REFERENCE

- **Nested-loop join:** a full inner scan for every outer row — O(n×m), correct by construction, no assumptions about key uniqueness needed
- **Hash join:** build a dict from one side, probe from the other — O(n+m), but only correct if the dict maps each key to a *list* of rows
- **Verified Finding 1:** both algorithms agree exactly on real customer/order data
- **Verified Finding 2:** a real, measured ≈44× speedup at 800×800 rows, with identical results
- **Verified Finding 3:** a naive overwrite-based hash join silently drops every duplicate-key row but the last one seen — no error, just fewer results
- **Verified Finding 4:** once the hash table is list-based, either side is safe to build on — the "build on the smaller table" rule is about performance, not correctness
- **Next chapter:** A Simple Query Planner — choosing an index and a join strategy automatically, verified against a naive always-scan baseline

CHAPTER 9 OF 10

A Simple Query Planner: Choosing an Index and a Join Strategy

Building a Database Engine: Transactions & Concurrency

Chapter 9 · A Simple Query Planner: Choosing an Index and a Join Strategy

Every chapter so far has left a choice to the caller: use an index or scan (Course 1, Chapter 9), nested-loop or hash join (Chapter 8, this course). A real planner makes that choice automatically, from real, measurable properties of the data — and, as this chapter finds out the hard way, the obvious way to estimate "which is cheaper" doesn't actually match reality until it's calibrated against real measurements.

The Obvious Cost Formula

```
def choose_join_strategy(n, m):
    nl_cost = n * m       # nested-loop: one comparison per (n, m) pair
    hj_cost = n + m       # hash join: one step per row, on each side
    return 'nested_loop' if nl_cost <= hj_cost else 'hash_join'
```

This predicts hash join wins as soon as `n=m=3` (`9 > 6`). Real measurement says otherwise.

Finding 1: The Naive Formula Is Wrong at Small Scale

VERIFIED DIRECTLY — REAL, AVERAGED TIMING (2,000 REPETITIONS PER SIZE, TO CUT THROUGH MICROSECOND-SCALE NOISE)

At `n=m=3`: nested-loop averages **0.47µs**, hash join **0.60µs** — nested-loop is actually faster, despite the formula predicting hash join. At `n=m=5`: the same mismatch. Pinpointing the real crossover precisely: nested-loop stays faster through `n=m=6`, and hash join only pulls ahead starting at `n=m=7`. The naive formula's own predicted crossover (`n=m=3`) misses the real one by a factor of more than 2×.

The reason: constructing a Python dict, hashing keys into it, and looking them up all carry real, non-trivial constant overhead the pure "count the operations" formula never accounts for. A handful of plain comparisons can genuinely be cheaper than building a hash table at all.

Finding 2: A Corrected, Empirically-Calibrated Formula

```
HASH_JOIN_OVERHEAD = 8   # calibrated against the real crossover measured above

def choose_join_strategy_corrected(n, m):
    nl_cost = n * m
    hj_cost = n + m + HASH_JOIN_OVERHEAD
    return 'nested_loop' if nl_cost <= hj_cost else 'hash_join'
```

VERIFIED DIRECTLY — EVERY SIZE, FROM N=M=2 THROUGH N=M=600, NOW MATCHES REAL BEHAVIOR

Re-checking every size tested — `2` through `600` — against the corrected formula: **every single prediction matches the real, measured winner**. A flat, real, measured constant folded into the cost estimate is enough to fix the small-scale mismatch entirely — for equal-sized inputs.

Finding 3: A Unified Planner, Measured Against a Naive Baseline

Combining Course 1's own index-vs-scan choice with this chapter's corrected join-strategy choice into one planner, and running it against a baseline that always scans and always uses nested-loop — on 1,500 customers and 3,000 orders.

VERIFIED DIRECTLY — A REAL, MEASURED ≈110× AGGREGATE SPEEDUP

300 point lookups: **0.71ms** with the index vs. **37.53ms** always scanning. One join: the planner correctly chose `hash_join`, running in **0.91ms** vs. **140.96ms** for the naive always-nested-loop baseline. Identical results both ways. Total: **1.62ms** vs. **178.49ms** — **≈110×** faster overall, with the planner never once choosing wrong.

An Honest Limitation: No Predicate Pushdown

"Find customer 42's own orders" can be answered two ways: join everything, then filter for customer 42 — or filter customers down to just that one row *first*, then join only that row against orders.

VERIFIED DIRECTLY — FILTERING FIRST IS DRAMATICALLY FASTER, AND THE PLANNER NEVER CONSIDERS IT

Join-then-filter on the full 1,500×3,000 tables (correctly using `hash_join`): **≈1ms**. Filter-then-join, using the index to find customer 42 first, then joining just that one row against orders: **≈0.1-0.2ms** — **5–9×** faster for this specific, highly selective query. This chapter's own planner never considers this option at all: `join_smart()` always joins the full tables and estimates cost purely from their sizes, with no concept of checking whether a WHERE clause could shrink one side down before the join even starts. Real query optimizers call this **predicate pushdown** — a genuinely bigger,

harder optimization than choosing an algorithm for a fixed order of operations, and a deliberate, honest scope boundary for this course's own simple planner.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Naive operation-count formula wrong at small scale	Maths for Programmers: Numerical Methods & Floating-Point Computation — the same lesson that a clean theoretical model and real measured behavior can diverge, and only measurement resolves which is right
Choosing index vs. scan	Building a Database Engine: Storage & Query Fundamentals Chapter 9 (Course 1) — `choose_where_plan()` is that exact function, reused unchanged
No predicate pushdown	An honest, deliberate scope boundary — a genuinely bigger project than this "simple" planner, matching Course 1's own honest "no cost-based optimizer beyond simple heuristics" scope statement from Chapter 1

Hands-On Exercises

EXERCISE 1

Test the corrected formula against a genuinely *asymmetric* shape — a tiny table (2 rows) joined against a much larger one (2,000, then 20,000, then 100,000 rows) — rather than the equal-sized pairs this chapter calibrated against. Confirm whether the formula's own prediction still matches real, measured behavior, and explain what you find.

View solution

EXERCISE 2

Run the same `WHERE id = 777` lookup through both `select()` (index-aware) and `select_naive()` (always scans) against the chapter's own 1,500-row customers table. Confirm both return the exact same single row, and explain why an index changing HOW an answer is found should never change WHAT the answer is.

View solution

EXERCISE 3

Run a range query (`WHERE id > 1495`) against the indexed `customers` table and confirm `choose_where_plan()` falls back to a full scan rather than using the index at all — even though the column has a real index built on it. Explain why, and what this planner would need to do differently to support it.

View solution

CHAPTER 9 QUICK REFERENCE

- **The naive formula:** `n*m` vs. `n+m` — reasonable in shape, wrong at small scale, verified reproducibly
- **Verified Finding 1:** real crossover is at n=m≈7, not n=m=3 as the naive formula predicts — dict construction has real constant overhead
- **Verified Finding 2:** a real, calibrated `+8` overhead term matches every equal-sized test from 2 to 600 rows
- **Verified Finding 3:** a unified index+join planner is ≈110× faster in aggregate than a naive always-scan/always-nested-loop baseline
- **Honest limitation:** no predicate pushdown — the planner is blind to filtering one side down before a join, missing a real 5–9× win for selective queries
- **Next chapter:** Capstone — a transactional, multi-table engine, bringing every chapter of this course together for the first time

CHAPTER 10 OF 10

Capstone — A Transactional, Multi-Table Engine

Building a Database Engine: Transactions & Concurrency

Chapter 10 · Capstone: A Transactional, Multi-Table Engine

Every chapter of this course built and verified one real piece in isolation: a write-ahead log, a recovery routine, undo-based rollback, locking, MVCC, foreign keys, two join algorithms, a cost-based planner. None of them were ever combined. This capstone wires as many of them together as genuinely can be combined into one real, disk-backed engine — and finds, as every capstone in this project's own "ambitious learning projects" tier has, that integration surfaces bugs no single chapter's own isolated tests could ever catch.

Step 1–3: Combining the WAL and Undo Logging for the First Time

Chapter 2's WAL and Chapter 4's undo log both wrap `write_page()` — but never once, anywhere in this course, on the same write. A `DurableTransactionalHeapFile` does both: logs the new page to the WAL (durability), records the old page in the undo log (atomicity), then applies the write.

VERIFIED DIRECTLY — A ROLLED-BACK WRITE COMES BACK FROM THE DEAD

A row is written, then a *business-logic failure* (not a crash) triggers `rollback()` — the row is correctly, genuinely gone from the real data file. Then, simulating an unrelated **later** crash, `recover()` is run. It **resurrects the rolled-back row** — the WAL still had a record of the original write, and recovery, having no idea it was ever undone, faithfully redoes it. Two mechanisms, each independently correct, produce a genuinely wrong combined result the moment they're used together.

THE FIX — ROLLBACK LOGS A COMPENSATING RECORD TOO

`rollback()` now appends the *restored* page content to the WAL as a new record, with a fresh LSN, instead of only updating the real file in place. Recovery, replaying the whole log in order — the original write, then the compensation — now lands on the same, correct, post-rollback state every time, crash or no crash. This is exactly ARIES's own compensation log record technique, arrived at independently by actually trying to combine two chapters that were each correct on their own.

Step 4: A Real Catalog, Durable and Rollback-Safe

Chapter 7's `Catalog` never used the WAL at all — it wrote straight to a plain `HeapFile`. Rebuilding it on top of the fixed, combined write path gives foreign keys real crash durability for the first time.

A GENUINE, DELIBERATE DEMONSTRATION — ROLLBACK UNDOES THE WHOLE TRANSACTION, NEVER JUST ONE STATEMENT

Two logically independent orders, sharing *one* transaction: the first is genuinely valid; the second violates a foreign key. Rolling back after the second one fails undoes **both** — including the first, perfectly valid order. This engine only supports transaction-level rollback, never statement-level rollback. Giving the two orders their own separate transactions resolves it correctly: the valid one commits and survives, the invalid one is rejected and rolled back alone.

Step 5: Real Concurrent Transactions, Protected by Locking

VERIFIED DIRECTLY — TWO REAL THREADS, ONE CORRECT FINAL ANSWER

Two real threads concurrently buy the same product, starting from 100 in stock — `30` and `40` units respectively — each holding Chapter 5's own exclusive lock across its entire read-modify-write cycle. Final stock: exactly `30` (`100 - 30 - 40`). The identical lock manager, wired into the real, disk-backed catalog for the first time.

Step 6–7: A Crash-and-Recover Cycle, and a Second Real Bug

Simulating the exact WAL-only crash from Chapter 2's own Finding 1 — log the write, never apply it — at the Catalog level, across three real tables sharing one WAL.

VERIFIED DIRECTLY — A SHARED WAL CORRUPTS DATA ACROSS TABLES

Recovery restores the missing order correctly — but the recovered orders table also shows **garbage rows** (implausible ids, enormous customer_id values), and Alice's own real order has **vanished entirely**. Root cause: `customers` , `products` , and `orders` all shared one WAL, and WAL records only ever store a bare `page_num` — with three separate heap files sharing one log, `page_num=0` for orders and `page_num=0` for products are genuinely ambiguous. Recovery replayed *every* record from every table onto the orders file alone, misinterpreting product and customer bytes as order rows.

THE FIX — ONE WAL PER TABLE, EXACTLY MATCHING CHAPTERS 2–3'S OWN ESTABLISHED PAIRING

Every table gets its own dedicated WAL file, never a shared one. Recovering orders' own WAL now only ever touches orders' own real records — Alice's order survives untouched, the crashed order is correctly restored, and no other table is ever at risk of corruption from another one's recovery.

Step 8: A Real Multi-Table Query, Planned Automatically

VERIFIED DIRECTLY — A REAL QUERY SPANNING THREE TABLES, TWO FOREIGN KEYS, AND A RECOVERED ROW

`customers` joined to `orders` (planner correctly chose `nested_loop` for this size), then matched against `products` — producing a complete, correct report, including the order that was recovered from the simulated crash in Step 7.

Every piece — the catalog, the FK constraints, the planner's own cost formula, the join algorithm — working together for the first time.

Step 9: An Honest Closing Note — MVCC Was Never Integrated

VERIFIED DIRECTLY — A REAL, COMMITTED MVCC TRANSACTION LEAVES ZERO BYTES ON DISK

A row is written and committed through `MVCCStore`, exactly as verified working in Chapter 6. Checking its own dedicated directory afterward: **no files at all**. Not a HeapFile, not a WAL — nothing. A real process restart would lose every row MVCC has ever held, in complete contrast to the WAL-backed catalog just verified surviving a real crash in Steps 6–7. Chapter 6 built and independently verified a genuine, working alternative to locking for isolation — but this course never connected it to the durable storage stack Chapters 1–4 spent so much effort building. Combining row-versioned MVCC with real, persistent storage is a substantial engineering project of its own — an honest, deliberate scope boundary to close this entire project on, not a gap to quietly paper over.

Chapter Attribution

CAPSTONE STEP	BUILT IN
Records, pages, heap files, B-tree indexes	Storage & Query Fundamentals Chapters 2–6 (Course 1)
Write-ahead logging	Chapter 2 (this course)
Crash recovery / replay	Chapter 3 (this course)
Undo logging & rollback	Chapter 4 (this course)
Locking (shared/exclusive)	Chapter 5 (this course)
MVCC (verified independently, never integrated)	Chapter 6 (this course)
Catalog, foreign keys	Chapter 7 (this course)
Nested-loop & hash join	Chapter 8 (this course)
Cost-based query planner	Chapter 9 (this course)

What This Project Doesn't Cover

- No client-server networking protocol — everything runs as a local, in-process library (stated in Storage & Query Fundamentals Chapter 1, held throughout both courses)
- No full SQL — a genuine but deliberately small subset (`CREATE TABLE` / `INSERT` / `SELECT` / `WHERE`, no `UPDATE` / `DELETE` at the SQL layer)
- No distributed or replicated storage

- No predicate pushdown — the planner never filters one side of a join down before joining (Chapter 9's own honest limitation)
- No B-tree range scans — only exact-match lookups can use an index (a real limitation traced to `btree_search()`'s own single-key design)
- No MVCC-backed durable storage — a genuine, working alternative concurrency strategy that was never connected to the disk-backed engine (this chapter's own closing finding)
- No statement-level rollback — only whole-transaction rollback (demonstrated directly in Step 4)
- No garbage collection for old MVCC versions, no vacuum, no free-page reclamation after a rollback-blanked page

Hands-On Exercises

EXERCISE 1

Simulate a crash affecting *two different tables* at the same moment — one logged-but-unapplied write to `customers`, one to `products` — using the fixed, per-table-WAL catalog. Recover each table from its own WAL and confirm neither table's own recovery affects the other's data at all.

View solution

EXERCISE 2

Re-run Step 5's own concurrent purchase test, but with the `LockManager` removed entirely. Confirm the outcome is genuinely wrong — either a silently incorrect final stock count, or an outright crash from two threads racing on the same shared file handle — and explain both possible failure modes.

View solution

EXERCISE 3

Commit a row through a real `MVCCStore`, then simulate a process restart by creating a brand-new `MVCCStore` object. Confirm the committed row is completely gone, and contrast this directly against the disk-backed catalog's own real recovery in Steps 6–7.

View solution

CAPSTONE QUICK REFERENCE

- **Bug 1 found:** rollback without a compensating WAL record lets a later crash-recovery resurrect a deliberately undone write — fixed by logging the restored page as a new WAL record

- **Bug 2 found:** a single shared WAL across multiple tables corrupts data during recovery, since page_num alone is ambiguous — fixed with one WAL per table
- **Genuine scope demonstration:** rollback always undoes the whole transaction, never a single failed statement
- **Verified end to end:** real concurrent locking, a real crash-and-recover cycle, and a real multi-table join, all through one integrated system
- **Honest closing finding:** MVCC (Chapter 6) was built and verified as a genuine alternative to locking, but was never connected to durable, disk-backed storage — a real, deliberate scope boundary, not a gap to hide
- **The project:** 20 chapters across two courses — Storage & Query Fundamentals and Transactions & Concurrency — building a real database engine from raw bytes to a working, multi-table, transactional, concurrent, query-planned system