

COMPANION VOLUME • STAGE 08 OF THE ENGINEER'S SYLLABUS

The Engineer's Ledger

A ledger is the oldest database there is: a book you only ever add to, kept so carefully that a business trusts it over anyone's memory. Every serious database still has one at its core, the write-ahead log, and everything else in it, the pages, the indexes, the planner, the locks, exists to make that record fast to search and safe to share. You already know how to ask a database questions. This volume is about what it does with them.

CHAPTERS	CODE EXAMPLES	EXERCISES	STAGE BUDGET
20	22	60	180 h

Contents

How to use this volume	4
Part I · The model	5
Chapter 01 · What a database promises	6
Chapter 02 · Keys, constraints and normal forms	12
Chapter 03 · Modelling the hard shapes	18
Chapter 04 · Objects meet tables: ORMs and the N+1 query	23
Part II · Storage	28
Chapter 05 · Pages, records and the buffer pool	29
Chapter 06 · B-trees	36
Chapter 07 · LSM trees and the price of fast writes	42
Part III · Queries	48
Chapter 08 · Indexes the planner will use	49
Chapter 09 · How a query executes: operators and joins	55
Chapter 10 · The planner and its statistics	61
Chapter 11 · A million rows under fifty milliseconds	66
Part IV · Transactions	72
Chapter 12 · What ACID really promises	73
Chapter 13 · Isolation levels and their anomalies	78
Chapter 14 · Locks, MVCC and deadlocks	84
Chapter 15 · The write-ahead log and crash recovery	90
Part V · Running it	97
Chapter 16 · Replication, backups and recovery points	98
Chapter 17 · Changing the schema under live traffic	104
Part VI · Beyond the relational core	109
Chapter 18 · NoSQL: choosing by access pattern	110
Chapter 19 · Caches and search indexes beside the database	116

Chapter 20 · Analytical data: columns, stars and pipelines	122
The twenty rules	128
Reading list	130

The chapters follow the shape of the rest of the series: the idea in one sentence, worked examples, the traps, then exercises with folded solutions, a checklist and further reading. Run the examples. A database is a machine you understand by watching it work, and every example here prints what it did.

This volume assumes you can write SQL. Joins, subqueries, CTEs, window functions and the interview patterns are taught in *SELECT Again*, and this volume starts where that course stops: what the database does with your query once it has it. You will meet SQL on every page, but you won't be taught it. The other prerequisite is a little computer architecture (stage 04): what a page of memory is, why a disk read costs so much more than a memory read, and what a cache does.

The examples run on a real database, or build a small working one. Python's standard library ships SQLite, so where an example needs a real engine it uses one: real query plans, real constraints, real locks, and a real million-row table. Where an example needs to show the inside of an engine, a B-tree splitting a page or a log being replayed after a crash, it builds a small version in plain Python. Every example was run before publication, and the output printed under it is what it actually printed.

Engines differ; the mechanisms don't. PostgreSQL, MySQL, SQLite and the rest make different choices: different default isolation levels, different index structures, different ways of keeping old versions of rows. This volume names those differences where they bite, but it teaches the mechanisms underneath, because those are what transfer. The stage's deliverable tests both halves. First, build a key-value store with a write-ahead log, crash recovery and a B-tree or LSM index (chapters 5 to 7 and 15 build the parts). Then take a slow query over a million rows and bring it under fifty milliseconds, documenting every plan change (chapter 11 does exactly that).

THE ONE QUESTION

When a query is slow or a write is unsafe, the instinct is to reach for a setting, an index or a different database. Most of the time the answer is already visible if you know where to look.

So before you change anything, ask: **what does the database have to read to answer this, and what does it have to write to the log before it can say "done"?** Reads are measured in pages, and a query that touches a million pages to return ten rows is slow no matter how fast the server is. Writes are measured in log records and the locks held while they're made. Every chapter in this book is a way of answering that question more precisely.

PART I

The model

Four chapters on what you store before how it's stored. What a database promises that a file can't, the keys and constraints that let it refuse bad data, the designs for the shapes that don't fit neatly into tables, and the ORM layer that sits between your objects and its rows.

What a database promises

3 exercises

A file is a place to put data; a database is a set of promises about it. It promises that a change happens completely or not at all, even if the power fails halfway. It promises that many people can change data at once without losing each other's work. And it promises that you can find any part of it without reading all of it. Every chapter in this volume is one of those promises and the machinery that keeps it.

Three promises a file can't keep

Most programs start by saving data to a file, and for a while that works. Then three things happen. The machine loses power while the file is being written, and the next start finds half a file. Two requests arrive at the same moment, both read the file, both change it, and one change disappears. And the file grows until finding one record means reading all of them. Those are the three promises a database exists to keep: **atomic, durable writes, safe concurrency, and finding without scanning**.

EXAMPLE promises.py · a file and a database under the same two accidents

```
import json, os, sqlite3, tempfile

d = tempfile.mkdtemp()
path = os.path.join(d, "accounts.json")

def save_in_place(data, crash_after=None):
    blob = json.dumps(data).encode()
    with open(path, "wb") as f:
        f.write(blob if crash_after is None else blob[:crash_after]) # opening for write truncates the old file
                                                                    # power cut mid-write

def save_atomically(data, crash_after=None):
    blob = json.dumps(data).encode()
    tmp = path + ".tmp"
    with open(tmp, "wb") as f:
        f.write(blob if crash_after is None else blob[:crash_after])
    if crash_after is not None:
        return # crashed before the rename: old file untouched
    f.flush(); os.fsync(f.fileno()) # make the new bytes durable first
    os.replace(tmp, path)           # atomic: readers see old or new, never half

def load():
    try:
        with open(path) as f:
            return json.load(f)
    except json.JSONDecodeError as e:
        return f"CORRUPT ({e.msg})"

# Promise 1: a write is all-or-nothing, even if the power fails halfway.
save_in_place({"alice": 100, "bob": 50})
save_in_place({"alice": 70, "bob": 80}, crash_after=12)
print("in-place write, crash halfway:", load())
```

```

save_atomically({"alice": 100, "bob": 50})
save_atomically({"alice": 70, "bob": 80}, crash_after=12)
print("atomic rename, crash halfway: ", load())

# Promise 2: two clients can't silently overwrite each other.
save_atomically({"visits": 0})
a = load()["visits"]; b = load()["visits"] # both clients read 0
save_atomically({"visits": a + 1}); save_atomically({"visits": b + 1})
print("file, two increments:      ", load())

dbfile = os.path.join(d, "app.db")
setup = sqlite3.connect(dbfile)
setup.execute("CREATE TABLE counters(name TEXT PRIMARY KEY, n INTEGER)")
setup.execute("INSERT INTO counters VALUES ('visits', 0)"); setup.commit(); setup.close()
A = sqlite3.connect(dbfile, timeout=0, isolation_level=None)
B = sqlite3.connect(dbfile, timeout=0, isolation_level=None)
A.execute("BEGIN IMMEDIATE") # client A takes the write lock
A.execute("UPDATE counters SET n = n + 1 WHERE name = 'visits'")
try:
    B.execute("UPDATE counters SET n = n + 1 WHERE name = 'visits'")
except sqlite3.OperationalError as e:
    print("client B must wait:      ", e)
A.execute("COMMIT")
B.execute("UPDATE counters SET n = n + 1 WHERE name = 'visits'") # B retries after A commits
print("database, two increments:    ", B.execute("SELECT n FROM counters").fetchone()[0])

```

OUTPUT

```

in-place write, crash halfway: CORRUPT (Expecting ',', delimiter)
atomic rename, crash halfway:  {'alice': 100, 'bob': 50}
file, two increments:          {'visits': 1}
client B must wait:           database is locked
database, two increments:      2

```

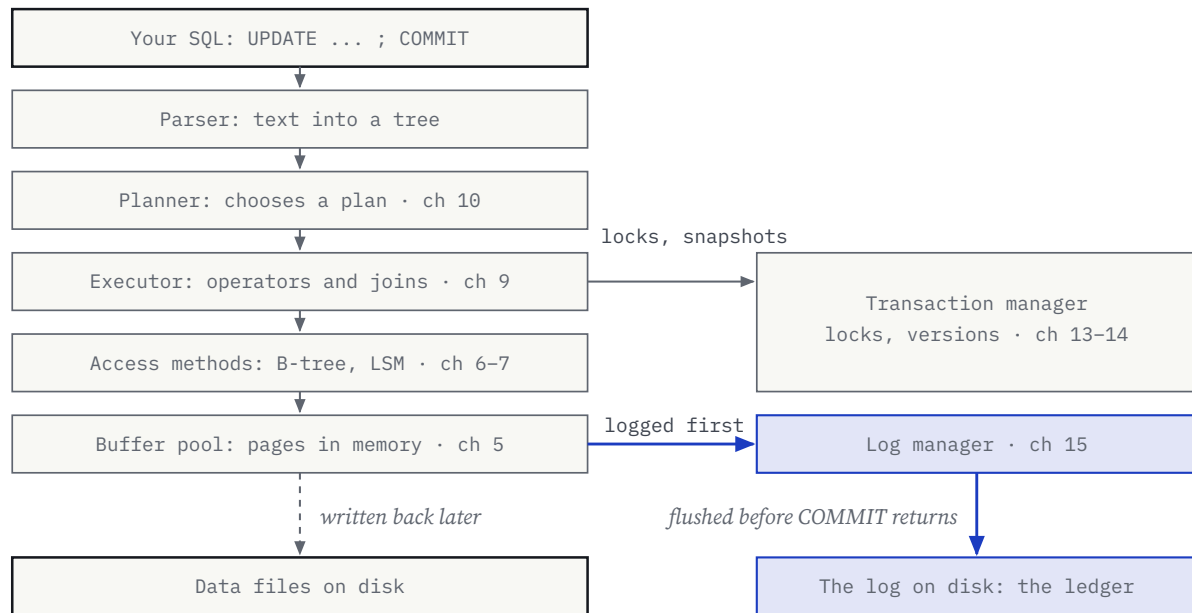
The first accident is the torn write. Opening a file for writing truncates it, so a crash between the truncate and the last byte leaves a file that is neither the old version nor the new one. The usual fix, writing a temporary file, flushing it to disk with `fsync` and renaming it over the original, does keep the old version safe, because a rename is atomic on POSIX file systems. But it only works for one file, it rewrites the whole file for every change, and it does nothing for the second accident.

The second accident is the lost update. Both clients read `0`, both write `1`, and two visits are recorded as one. Nothing crashed and nothing reported an error: the data is simply wrong. The database turns the same two increments into `2` because it won't let the second writer act on stale data. Client B is told to wait while A holds the write lock, then its update runs against A's committed value. That waiting is concurrency control, and a whole part of this book (chapters 12 to 15) is about doing it without making everyone wait all the time.

The third promise isn't in the example because it's the subject of Parts II and III: a database keeps its data in fixed-size pages and builds indexes over them, so that finding one customer among ten million means reading three or four pages instead of all of them.

What's inside

Almost every relational database has the same parts, arranged the same way. Your SQL is parsed into a tree, the planner turns the tree into a plan, and the executor runs the plan as a pipeline of operators. The operators read through access methods, B-trees and their relatives, which ask the buffer pool for pages, which reads them from disk only when they aren't already in memory. Alongside, a transaction manager decides who may see and change what, and a log manager records every change before it is allowed to count.



The parts of a relational database, with the chapter that covers each. The main path is the promise of durability: a change is written to the log, and the log is flushed to disk, before COMMIT returns. The changed data pages stay in memory and reach the data files later, which is safe because the log can rebuild them.

The detail worth noticing now is the two arrows to disk. When you commit, the database does not rush your changed rows into the data files. It appends a description of the change to the log and flushes the log, a fast sequential write, and only then tells you the transaction is done. The data pages are written back later, in the background, in whatever order is efficient. If the machine crashes in between, the log holds everything needed to redo the work. The log is the ledger, and the data files are a convenient, searchable copy of what it says. Chapter 15 builds one.

Choosing an engine is choosing trade-offs

Databases differ in where they sit and what they optimise. **Embedded** databases such as SQLite run inside your process and store everything in one file. They are the most widely deployed database engine there is, in every phone and browser, and the right default for local and single-writer data. **Client-server** databases such as PostgreSQL and MySQL run as their own service, accept many concurrent writers over the network, and add replication, roles and online maintenance. **Transactional** (OLTP) systems are built for many small reads

and writes; **analytical** (OLAP) systems are built for scans over billions of rows, and store data in columns instead of rows (chapter 20). **Non-relational** stores trade the query language for a particular access pattern or scale (chapter 18). None of these is better in general. The useful question is always which promises you need, and how much each one costs.

This volume uses SQLite for its real examples because it ships with Python, so every example runs anywhere without setup. Its mechanisms, pages, B-trees, a write-ahead log, locks, a cost-based planner, are the same ones the server databases use, at a smaller scale.

COMMON TRAPS

Using files where you need a database, then rediscovering torn writes and lost updates one incident at a time.

Renaming without `fsync`, so the rename survives a crash but the bytes it points at don't. Durable atomic replace needs a flush of the file and, strictly, of its directory too.

Assuming promises you never asked for. A database keeps them per transaction, and many client libraries run in autocommit mode, where every statement is its own transaction.

Confusing durability with backups. Durability survives a crash; it does nothing for a mistaken `DELETE` or a lost disk (chapter 16).

Choosing an engine by popularity rather than by the promises and the access patterns you need.

EXERCISES

EXERCISE 1.1 Which promise broke?

Name the promise each incident breaks.

- After a power cut, the app's settings file is empty and the app won't start.
 - Two warehouse workers both see one item left, both sell it, and the stock count reads zero.
 - "Find customer by email" took 5 ms at launch and takes 4 seconds a year later.
-

SOLUTION

- a. **Atomic, durable writes.** The file was truncated and only partly rewritten when the power went. Write to a temporary file, flush it, and rename it over the original, or use a database.
- b. **Safe concurrency.** A lost update: both read `1` and both wrote `0`, so one sale was never subtracted. The fix is to make the read and the write one atomic step, for example `UPDATE stock SET n = n - 1 WHERE id = ? AND n > 0`, and check that one row changed.
- c. **Finding without scanning.** With no index on `email`, every lookup reads the whole table, and the table grew. An index turns the scan into a few page reads (chapters 6 and 8).

EXERCISE 1.2 The limits of the rename trick

The atomic rename protected the file in the example. Name three things it does not solve, and say which part of a database solves each.

SOLUTION

First, **concurrent writers**: two processes can each write a temporary file and rename it, and the second rename silently discards the first one's change, the same lost update as before. A database solves that with locks or version checks (chapters 13 and 14). Second, **changes that span more than one file**: debiting one account file and crediting another can't be made atomic with renames, because a crash between the two renames leaves money missing. A database makes the whole transaction atomic through its log (chapter 15). Third, **cost**: every change rewrites the whole file, so a one-byte update to a gigabyte of data writes a gigabyte. A database writes a small log record and updates one page in memory. A fourth, if you found it: the rename is only durable once the directory entry is flushed too, which the example glosses over.

EXERCISE 1.3 What is on disk when COMMIT returns?

Using the diagram, trace `UPDATE accounts SET balance = balance - 30 WHERE id = 7; COMMIT;`. When `COMMIT` returns, which of these are guaranteed to be on disk: the log record for the change, the changed data page, or the index page for `id = 7`? What happens to the others if the power fails one second later?

SOLUTION

Only the **log record** (and the commit record after it) is guaranteed to be on disk. That is the whole point of the write-ahead log: a sequential append plus one flush is cheap, so the database pays for it on every commit. The changed data page is in the buffer pool and may not reach the data file for seconds or minutes. The index page is untouched here, because `balance` isn't indexed; if it were, its change would be logged too. If the power fails one second later, recovery reads the log, finds the committed change whose page never made it to disk, and redoes it. A committed transaction is never lost, and the data files catch up from the log.

CHECKLIST

- ☐ Can you name the three promises and the incident that breaks each?
- ☐ Do you know why an atomic rename isn't a substitute for a transaction?
- ☐ Can you trace a query and a commit through the parts of a database?
- ☐ Do you know what is guaranteed to be on disk when COMMIT returns?
- ☐ Can you say which promises your application actually needs?

GO DEEPER

PAPER [Architecture of a Database System](#) Hellerstein, Stonebraker & Hamilton

REFERENCE [Atomic Commit in SQLite](#) SQLite documentation

ESSAY [Files are hard](#) Dan Luu

Keys, constraints and normal forms

3 exercises

The cheapest bug fix is the row the database refused to store. Keys and constraints let the database enforce the facts your code assumes, for every client and every code path at once. Normalisation is the discipline of storing each fact exactly once, so that the data can't disagree with itself. Denormalise later, on purpose, with a plan for keeping the copies honest.

Constraints are rules nobody can skip

Every application has rules about its data: emails are unique, an order belongs to a customer who exists, a total is never negative. You can check them in application code, and you should, to give users friendly errors. But application checks only cover the code paths that remember to run them. They miss the admin script, the batch import, the second service written by another team next year, and, as exercise 2.2 shows, two requests racing each other. A constraint declared in the schema is checked by the database on every write, from every client, inside the same transaction as the write. It is the only check that can't be bypassed.

EXAMPLE constraints.py · five bad writes, five refusals, and one trap

```
import sqlite3

db = sqlite3.connect(":memory:", isolation_level=None)  # autocommit: each statement stands alone
db.execute("PRAGMA foreign_keys = ON")                 # SQLite leaves this OFF unless you ask
db.executescript("""
CREATE TABLE customers(
    id      INTEGER PRIMARY KEY,
    email TEXT NOT NULL UNIQUE CHECK (email LIKE '%_@_%'));
CREATE TABLE orders(
    id          INTEGER PRIMARY KEY,
    customer_id INTEGER NOT NULL REFERENCES customers(id) ON DELETE RESTRICT,
    total_cents INTEGER NOT NULL CHECK (total_cents >= 0));
INSERT INTO customers VALUES (1, 'ana@example.com');
INSERT INTO orders VALUES (10, 1, 2500);
""")
attempts = [
    ("duplicate email",          "INSERT INTO customers VALUES (2, 'ana@example.com')"),
    ("malformed email",         "INSERT INTO customers VALUES (3, 'not-an-email')"),
    ("order for a missing customer", "INSERT INTO orders VALUES (11, 99, 100)"),
    ("negative total",           "INSERT INTO orders VALUES (12, 1, -5)"),
    ("delete a customer with orders", "DELETE FROM customers WHERE id = 1"),
]
for label, sql in attempts:
    try:
        db.execute(sql)
        print(f"ACCEPTED {label}")
    except sqlite3.IntegrityError as e:
        print(f"rejected {label:30} {e}")
```

```
db.execute("PRAGMA foreign_keys = OFF")
db.execute("INSERT INTO orders VALUES (13, 99, 100)")
print("with foreign_keys OFF, an order for customer 99 is quietly ACCEPTED")
```

OUTPUT

```
rejected duplicate email          UNIQUE constraint failed: customers.email
rejected malformed email         CHECK constraint failed: email LIKE '%_@_%'
rejected order for a missing customer FOREIGN KEY constraint failed
rejected negative total          CHECK constraint failed: total_cents >= 0
rejected delete a customer with orders FOREIGN KEY constraint failed
with foreign_keys OFF, an order for customer 99 is quietly ACCEPTED
```

Each refusal names the rule that stopped it. `UNIQUE` blocks the duplicate email, `CHECK` rejects the malformed email and the negative total, and the foreign key blocks both the order for a customer who doesn't exist and the deletion of a customer who still has orders. `ON DELETE RESTRICT` is a choice: `CASCADE` would delete the orders too, and `SET NULL` would orphan them on purpose. Pick the one that matches what deleting a customer means in your business.

The last line is the trap. SQLite, for backward compatibility, enforces foreign keys only when the connection turns them on, and turning them on (or off) silently does nothing inside an open transaction. MySQL's old MyISAM engine accepted foreign key syntax and ignored it entirely. Never assume a constraint is enforced: write a test that tries to violate it.

Keys

A **key** is a set of columns whose values identify one row. A table can have several **candidate keys** (a customer might be identified by an internal id and also by an email); one is chosen as the **primary key** and the others should still be declared `UNIQUE`, or the database won't know they are keys. A **surrogate key**, an auto-incrementing integer or a UUID, carries no meaning and never needs to change; a **natural key**, like an ISBN, carries meaning and sometimes does change. SELECT Again's schema-design module covers that choice. The rule that matters here is simpler: every fact you believe is unique deserves a `UNIQUE` constraint, whichever key you pick.

One fact, one place

Normalisation sounds academic and is entirely practical. When the same fact is stored in many rows, an update that changes some copies and misses others leaves the database contradicting itself, and no query can tell you which copy is right. The tool for spotting this is the **functional dependency**: $X \rightarrow Y$ means that knowing `X` tells you `Y`. An order id tells you the customer; a customer id tells you the email. A table is in good shape, formally **Boyce-Codd normal form**, when the left side of every dependency is a key. When it isn't, that dependency's facts repeat.

EXAMPLE `normal.py` · find the keys and the repeated facts in a wide table

```

from itertools import combinations

def closure(attrs, fds):
    """Everything the attributes determine, following dependencies until nothing changes."""
    result, changed = set(attrs), True
    while changed:
        changed = False
        for lhs, rhs in fds:
            if lhs <= result and not rhs <= result:
                result |= rhs; changed = True
    return result

def candidate_keys(attrs, fds):
    keys = []
    for size in range(1, len(attrs) + 1):
        for combo in combinations(sorted(attrs), size):
            c = set(combo)
            if closure(c, fds) == attrs and not any(k <= c for k in keys):
                keys.append(c)
    return keys

def bcnf_violations(attrs, fds):
    """A dependency whose left side is not a key means one fact is stored many times."""
    return [(lhs, rhs) for lhs, rhs in fds if closure(lhs, fds) != attrs]

# One wide table, as spreadsheets and first drafts love to do it.
A = set("order_id product_id qty customer_id customer_email product_name unit_price".split())
F = [{("order_id", "product_id", {"qty"}),
      {"order_id", {"customer_id"}},
      {"customer_id", {"customer_email"}},
      {"product_id", {"product_name", "unit_price"}}]

print("candidate keys:", [sorted(k) for k in candidate_keys(A, F)])
for lhs, rhs in bcnf_violations(A, F):
    print(f"  {' '.join(sorted(lhs)):11} -> {' '.join(sorted(rhs)):26} repeats on every row
          that shares {' '.join(sorted(lhs))}")
print("split each violation into its own table:")
for lhs, rhs in F:
    print(f"  table keyed by ({' '.join(sorted(lhs))}) holding {' '.join(sorted(rhs))}")

```

OUTPUT

```

candidate keys: [['order_id', 'product_id']]
order_id    -> customer_id           repeats on every row that shares order_id
customer_id -> customer_email        repeats on every row that shares customer_id
product_id  -> product_name, unit_price repeats on every row that shares product_id
split each violation into its own table:
table keyed by (order_id, product_id) holding qty
table keyed by (order_id) holding customer_id
table keyed by (customer_id) holding customer_email
table keyed by (product_id) holding product_name, unit_price

```

The wide table's only key is `(order_id, product_id)`, one row per line item. Every other dependency has a smaller left side, and each one means a fact repeated across rows. The customer's email is copied onto every line of every order they place, so changing it means updating hundreds of rows and getting all of them. The product name is copied onto every line that ever sold it. The classic names describe the two shapes: `order_id → customer_id`

depends on part of the key (a **second normal form** violation), and `customer_id` → `customer_email` depends on a non-key column (a **third normal form** violation). The fix is the same for both: give each dependency a table of its own, keyed by its left side.

The output hides one deliberate trap. The tool treats `unit_price` as a fact about the product, and if it means the product's current price, it is. But on an order line, the price usually means *the price this customer paid*, which must not change when the catalogue price does. That's a different fact with a different key: it belongs on the order line. Normalisation can only be as right as the dependencies you write down, so the real work is asking what each column actually means.

Denormalising on purpose

A normalised schema is the right default for data you write. Sometimes you deliberately copy a fact anyway: an order count kept on the customer row so a list page doesn't count orders on every view, a materialised summary for a dashboard, a search document that flattens five tables. That's fine if you treat the copy as derived data, with one owner and a way to rebuild it. Update the copy in the same transaction as the source, or with a trigger, or rebuild it on a schedule and accept the lag, and keep a reconciliation query that finds copies that have drifted. The failure mode is the copy nobody owns, updated by some code paths and not others, quietly diverging for a year.

COMMON TRAPS

Rules enforced only in application code, which the next script, service or race walks straight past.

Constraints that silently aren't on: SQLite foreign keys without the pragma, engines that parse `CHECK` and ignore it.

Forgetting that `UNIQUE` allows many NULLs in most databases, because NULL never equals NULL.

Declaring only the primary key and leaving the other candidate keys unenforced.

Denormalising without an owner or a reconciliation check for the copies.

EXERCISES

EXERCISE 2.1 Find the repeated facts

A timesheet table has columns (emp_id, emp_name, dept_id, dept_name, project_id, hours), one row per employee per project. Write down the functional dependencies, the key, and the tables you'd split it into.

SOLUTION

Dependencies: emp_id → emp_name, dept_id; dept_id → dept_name; emp_id, project_id → hours. The key is (emp_id, project_id). emp_id → emp_name, dept_id depends on part of the key, so an employee's name and department repeat on every project row. dept_id → dept_name is transitive, so a department's name repeats for every employee in it. Split into employees(emp_id, emp_name, dept_id), departments(dept_id, dept_name) and timesheet(emp_id, project_id, hours), with foreign keys from employees.dept_id and timesheet.emp_id. Renaming a department is now one update to one row.

EXERCISE 2.2 The check that races

A sign-up handler runs SELECT 1 FROM users WHERE email = ? and inserts the user only if no row comes back. There's no UNIQUE constraint. Explain how duplicate accounts still appear, and what the fix is.

SOLUTION

Two sign-ups with the same email arrive together (a double-clicked button is enough). Both run the SELECT before either has inserted, both see no row, and both insert. Check-then-act in application code is a race, and wrapping it in a transaction doesn't help at the default isolation level of most databases, because neither transaction can see the other's uncommitted insert (chapter 13). The fix is a UNIQUE constraint on email, ideally on a normalised form such as lower case: the database serialises the two inserts, the second fails with a constraint violation, and the handler turns that error into a friendly "this email is already registered". Keep the SELECT for a good user experience if you like, but the constraint is the check that actually holds.

EXERCISE 2.3 Is this a copy?

A reviewer flags order_lines.product_name and order_lines.unit_price as normalisation violations, because both are in products. Are they right about both columns?

SOLUTION

It depends on what each column means, which is the real lesson.

`unit_price` on an order line is almost always the price paid at the time of sale. That's a fact about the line, keyed by the line, and it must stay fixed when the catalogue price changes, so it isn't a violation.

`product_name` is less clear. If it's there so that invoices show the name the customer saw at the time, it's also a historical fact about the line and should stay. If it's there only to save a join, it's a copy of `products.name` that will drift when a product is renamed, and the reviewer is right. Decide which it is, write it down in a comment on the column, and if it's a copy, give it an owner or remove it.

CHECKLIST

- ☐ Is every rule your code assumes also a constraint in the schema?
- ☐ Have you tested that each constraint is actually enforced?
- ☐ Is every candidate key declared, not just the primary key?
- ☐ Can you find the functional dependencies of a table and spot the repeated facts?
- ☐ Does every denormalised copy have an owner and a reconciliation check?

GO DEEPER

PAPER [A Relational Model of Data for Large Shared Data Banks](#) E. F. Codd, 1970

COURSE [Databases: relational design](#) Jennifer Widom, Stanford

REFERENCE [SQLite foreign key support](#) SQLite documentation

Modelling the hard shapes

3 exercises

Most schemas are easy until they have to hold a tree, a history, an amount of money, or a thing that can be several kinds of thing. Each of those shapes has a handful of known designs with known costs. Picking one deliberately, on day one, is far cheaper than discovering its trade-off in production with ten million rows already written.

Trees

Categories, comment threads, org charts and folders are all trees, and a table is flat. The obvious design is the **adjacency list**: each row stores its parent's id. It's easy to write and to move a subtree (change one `parent_id`), but reading a whole subtree needs a recursive query that walks one level at a time. The **closure table** stores every ancestor-descendant pair, with its depth, in a second table. Reading a subtree, all ancestors, or the depth of any node becomes one indexed lookup, at the cost of more rows and more work when the tree changes. Two other designs are worth knowing by name: the **materialised path** stores `'/1/2/3/'` on each row and finds a subtree with a prefix search, and **nested sets** store left and right numbers that make subtree reads fast and every insert expensive.

EXAMPLE shapes.py · one tree two ways, and why money isn't a float

```
import sqlite3

db = sqlite3.connect(":memory:")
db.executescript("""
CREATE TABLE category(id INTEGER PRIMARY KEY, parent_id INTEGER REFERENCES category(id), name
TEXT);
INSERT INTO category VALUES
(1, NULL, 'Electronics'), (2, 1, 'Computers'), (3, 2, 'Laptops'), (4, 2, 'Desktops'),
(5, 1, 'Audio'), (6, 5, 'Headphones'), (7, 3, 'Gaming laptops');
""")

# Shape 1: adjacency list. Simple to write; reading a subtree needs a recursive query.
subtree_cte = db.execute("""
WITH RECURSIVE sub(id, name, depth) AS (
SELECT id, name, 0 FROM category WHERE id = 2
UNION ALL
SELECT c.id, c.name, sub.depth + 1 FROM category c JOIN sub ON c.parent_id = sub.id)
SELECT name, depth FROM sub ORDER BY depth, name""").fetchall()

# Shape 2: closure table. Store every ancestor-descendant pair; a subtree is one indexed
lookup.
db.executescript("""
CREATE TABLE category_path(ancestor INTEGER, descendant INTEGER, depth INTEGER,
PRIMARY KEY (ancestor, descendant));
WITH RECURSIVE p(a, d, depth) AS (
SELECT id, id, 0 FROM category
UNION ALL
SELECT p.a, c.id, p.depth + 1 FROM p JOIN category c ON c.parent_id = p.d)
INSERT INTO category_path SELECT a, d, depth FROM p;
```

```

"""
subtree_closure = db.execute("""
    SELECT c.name, p.depth FROM category_path p JOIN category c ON c.id = p.descendant
    WHERE p.ancestor = 2 ORDER BY p.depth, c.name""").fetchall()

print("subtree of Computers:", subtree_cte)
print("same answer from the closure table:", subtree_cte == subtree_closure)
print("rows: category", db.execute("SELECT COUNT(*) FROM category").fetchone()[0],
      "| category_path", db.execute("SELECT COUNT(*) FROM category_path").fetchone()[0])

# Money: binary floats cannot represent most decimal fractions exactly.
total_float, total_cents = 0.0, 0
for _ in range(1000):          # a running balance, the way application code adds
    total_float += 19.99
    total_cents += 1999
print("1000 x 19.99 as float:", total_float)
print("1000 x 1999 in cents: ", total_cents, "=", f"{total_cents // 100}.{total_cents % 100:02d}")
print("round(2.675, 2) ->", round(2.675, 2), " (2.675 is stored as 2.67499999...)")

```

OUTPUT

```

subtree of Computers: [('Computers', 0), ('Desktops', 1), ('Laptops', 1), ('Gaming laptops',
2)]
same answer from the closure table: True
rows: category 7 | category_path 18
1000 x 19.99 as float: 19990.000000000135
1000 x 1999 in cents: 1999000 = 19990.00
round(2.675, 2) -> 2.67 (2.675 is stored as 2.67499999...)

```

Both tree designs return the same subtree. The closure table pays for it in storage, 18 path rows for 7 categories, because every node is also paired with itself and with each of its ancestors. In return, its subtree query has no recursion, and its `(ancestor, descendant)` primary key is an index the planner can use directly. Choose by the workload: an adjacency list for trees that change often and are read a level at a time, a closure table or materialised path for trees read whole far more often than they change.

Money

The last lines of the example are the reason money is never a binary float. A float stores most decimal fractions as the nearest binary fraction, so a running total of 1,000 prices of 19.99 drifts off by a few billionths. Rounding 2.675 to two places gives 2.67, because 2.675 is actually stored as 2.67499999.... Each error is tiny, and an accounting system that is off by a cent in a few thousand invoices fails its audit anyway. Store money either as an integer number of the currency's minor unit (cents, pence) or as an exact `DECIMAL(19, 4)`. Always store the currency next to the amount, and decide explicitly where rounding happens, because tax and currency conversion force it somewhere.

Things that are several kinds of thing

A payment might be a card payment, a bank transfer or a voucher, each with its own fields. **Single-table inheritance** puts them all in one table with a `type` column and nullable columns for every kind: simple to query, but the database can't enforce that a card payment has a card number. **Class-table inheritance** keeps the shared columns in a `payments` table

and each kind's columns in its own table, keyed by the payment id: the constraints work, at the cost of a join. The design to avoid is the **polymorphic foreign key**, a pair like `(commentable_type, commentable_id)` that might point at a post or a photo. It can't be a real foreign key, so the database can no longer guarantee that the thing it points at exists. Where you need that shape, use one nullable foreign key column per target, with a `CHECK` that exactly one is set.

Time and history

Store timestamps in UTC, as a timestamp type rather than text, and convert to local time only for display. Daylight-saving shifts make local times ambiguous for one hour every autumn. When you need history, don't overwrite and hope: give the row `valid_from` and `valid_to` columns so you can ask what was true on a given date, or append changes to an event or audit table so you can ask what happened. **Soft deletes**, a `deleted_at` column instead of a real delete, preserve history but cost more than they look. Every query must remember to filter them, and unique constraints break, because the deleted row still holds the email a new user wants (exercise 3.3).

Semi-structured data

Modern relational databases store JSON well, and a JSON column is the right home for data whose shape genuinely varies: a webhook payload you keep for debugging, product attributes that differ by category, user preferences. It's the wrong home for data you filter, join or constrain, because the schema can no longer enforce types, keys or references inside it, and indexing it takes expression indexes you must remember to create. A useful rule: any field you query by gets promoted to a real column.

COMMON TRAPS

Polymorphic foreign keys, which give up referential integrity for convenience.

Money as a float, or an amount stored without its currency.

Local timestamps, ambiguous for an hour every year and wrong across time zones.

Soft deletes everywhere, with queries that forget the filter and unique constraints that block re-registration.

JSON as an escape from design: fields you filter on, hidden where the schema can't check them.

EXERCISES

EXERCISE 3.1 Pick a tree design

Choose a design for each and say why: (a) comment threads, read whole on every page view and almost never moved; (b) an org chart that is reorganised every quarter; (c) a file system whose most common query is "everything under this folder".

SOLUTION

- A **closure table** or **materialised path**. Reads dominate overwhelmingly and moves are rare, so pay at write time to make "the whole thread" one indexed query.
- An **adjacency list**. Moving a team is one `parent_id` update; in a closure table it means deleting and reinserting every path through the moved subtree. The recursive query for a subtree is fine at org-chart sizes.
- A **materialised path** such as `/home/ana/projects/`. "Everything under this folder" becomes a prefix search that a normal B-tree index serves directly, and the path is also what users see. Renaming a folder rewrites the paths beneath it, an acceptable cost for a rare operation.

EXERCISE 3.2 What did it cost on March 3rd?

Prices change over time, and support needs to answer "what was this product's price on a given date". Design the table and the query, and say how to stop two prices being valid at the same moment.

SOLUTION

Use `product_prices(product_id, price_cents, currency, valid_from, valid_to)`, with `valid_to` NULL for the current price, and treat the range as half-open, `[valid_from, valid_to)`, so that consecutive prices meet without overlapping. The query is `WHERE product_id = ? AND valid_from <= :d AND (valid_to IS NULL OR valid_to > :d)`, served by an index on `(product_id, valid_from)`. To prevent overlaps, PostgreSQL can enforce it with an exclusion constraint on a range column (`EXCLUDE USING gist (product_id WITH =, valid_period WITH &&)`). Elsewhere, change prices only through one transaction that closes the old row and opens the new one, with a uniqueness rule such as one open row per product.

EXERCISE 3.3 Soft deletes and unique emails

Users are soft-deleted with `deleted_at`, and `email` is `UNIQUE`. A deleted user's email can now never be registered again. Fix it without dropping the uniqueness rule.

SOLUTION

Make the uniqueness apply only to live rows, with a partial unique index: `CREATE UNIQUE INDEX users_live_email ON users(email) WHERE deleted_at IS NULL`, supported by PostgreSQL and SQLite. Any number of deleted rows may share an email, but only one live row can hold it. In MySQL, which lacks partial indexes, the usual workaround is a generated column that is the email for live rows and NULL for deleted ones, made unique. The broader lesson: before adopting soft deletes, list the constraints and queries that will need to know about them, and consider moving deleted rows to an archive table instead.

CHECKLIST

- ☐ Did you choose a tree design by its read and write pattern?
- ☐ Is money stored as integer minor units or exact decimals, with its currency?
- ☐ Are timestamps UTC, stored as timestamps?
- ☐ Does every reference to another row use a real foreign key?
- ☐ Are fields you query by real columns rather than keys inside JSON?

GO DEEPER

BOOK [SQL Antipatterns](#) Bill Karwin

REFERENCE [Range types and exclusion constraints](#) PostgreSQL documentation

REFERENCE [Class table inheritance](#) Martin Fowler

Objects meet tables: ORMs and the N+1 query

3 exercises

An object-relational mapper turns rows into objects and saves you hundreds of lines of plumbing, but it hides the one number you most need to see: how many queries your code sends. Use the ORM for the many simple reads and writes it handles well, count queries the way you'd assert on any other output, and write SQL where the object model and the relational model disagree.

Two models of the same data

Your code thinks in objects: an author has a list of books, you follow references from one object to the next, and identity means "the same object in memory". The database thinks in relations: sets of rows, related by matching values, combined by joins, where identity means "the same key". The gap between the two is called the **object-relational mismatch**, and an ORM such as Django's, SQLAlchemy, Hibernate or ActiveRecord bridges it. It does a lot well. It maps rows to objects and back, always parameterises queries (which prevents SQL injection), tracks which objects changed and writes them in one transaction, and generates migrations. The trouble is that following a reference in object code looks free, and in an ORM it can be a network round trip.

EXAMPLE `nplus1.py` · the same page three ways, counting round trips

```
import sqlite3

class Connection:
    """Wraps a database connection: counts round trips and charges each one network latency."""
    def __init__(self, conn, rtt_ms=1.0):
        self.conn, self.rtt_ms, self.queries = conn, rtt_ms, 0
    def query(self, sql, params=()):
        self.queries += 1
        return self.conn.execute(sql, params).fetchall()
    def latency_ms(self):
        return self.queries * self.rtt_ms

raw = sqlite3.connect(":memory:")
raw.execute("CREATE TABLE author(id INTEGER PRIMARY KEY, name TEXT)")
raw.execute("CREATE TABLE book(id INTEGER PRIMARY KEY, author_id INTEGER, title TEXT)")
raw.execute("CREATE INDEX book_author ON book(author_id)")
for a in range(1, 101):
    raw.execute("INSERT INTO author VALUES (?, ?)", (a, f"Author {a}"))
    raw.executemany("INSERT INTO book(author_id, title) VALUES (?, ?)",
                    [(a, f"Book {a}.{i}") for i in range(3)])

class Author:
    """What a lazy-loading ORM gives you: an object whose books load on first access."""
    def __init__(self, conn, row):
        self._conn, (self.id, self.name), self._books = conn, row, None
```

```

@property
def books(self):
    if self._books is None:
        # looks like an attribute, is a query
        self._books = self._conn.query("SELECT title FROM book WHERE author_id = ?",
                                         (self.id,))
    return self._books

def page_lazy(c):
    # the innocent-looking loop in a template
    authors = [Author(c, r) for r in c.query("SELECT id, name FROM author")]
    return {a.name: len(a.books) for a in authors}

def page_join(c):
    # ask for exactly what the page shows, in one query
    return dict(c.query("""SELECT a.name, COUNT(b.id) FROM author a
                           LEFT JOIN book b ON b.author_id = a.id GROUP BY a.id"""))

def page_prefetch(c):
    # the ORM's "eager loading": one query for parents, one for all
    # children
    authors = c.query("SELECT id, name FROM author")
    ids = [a for a, _ in authors]
    books = c.query(f"SELECT author_id FROM book WHERE author_id IN ({','.join('? ' *
        len(ids))})", ids)
    counts = {}
    for (author_id,) in books:
        counts[author_id] = counts.get(author_id, 0) + 1
    return {name: counts.get(i, 0) for i, name in authors}

results = []
for fn in (page_lazy, page_join, page_prefetch):
    c = Connection(raw, rtt_ms=1.0)
    results.append(fn(c))
    print(f"{fn.__name__:14} {c.queries:4} queries ~{c.latency_ms():5.0f} ms of round trips at
        1 ms each")
print("same page every time:", results[0] == results[1] == results[2])

```

OUTPUT

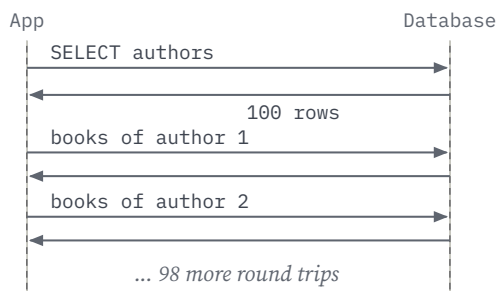
```

page_lazy      101 queries ~ 101 ms of round trips at 1 ms each
page_join      1 queries  ~  1 ms of round trips at 1 ms each
page_prefetch  2 queries  ~  2 ms of round trips at 1 ms each
same page every time: True

```

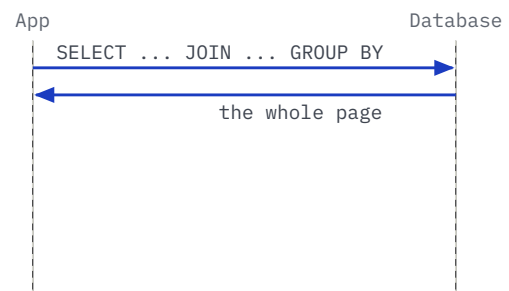
All three functions produce the same page. The lazy version sends **101 queries**: one for the authors, then one for each author's books the moment the loop touches `a.books`. This is the N+1 query problem, and it is invisible in the code, because `a.books` reads like an attribute. Each query is fast, so it doesn't show up in a profiler either. What shows up is the page: at one millisecond per round trip to a database on another machine, the round trips alone cost a tenth of a second, and on a page with a thousand rows, a full second.

Lazy loading



101 round trips $\approx$ 101 ms

One query with a join



1 round trip $\approx$ 1 ms

The N+1 problem is a latency problem, not a query-speed problem. Each lazy query is fast; there are just a hundred of them, and each one pays the network round trip. The join asks once for exactly what the page shows.

The fixes

There are two good fixes, and the example shows both. A **join** fetches parents and children in one query. For a count, as here, it's exact and minimal. The ORM's **eager loading** (`select_related` in Django, `joinedload` in SQLAlchemy) does the same for related objects. **Prefetching** fetches all the parents, then all their children in one `IN (...)` query, and stitches them together in memory: two queries regardless of N (`prefetch_related`, `selectinload`). Prefetching is often the better default for one-to-many relations, because a join repeats each parent's columns on every child row, and joining two collections at once multiplies them (exercise 4.2). Services built on GraphQL get the same effect from a **batching loader** that collects every id requested in one tick and fetches them together.

Other things the ORM hides

N+1 is the famous one, but not the only one. Loading full objects to change one field reads every column and writes them all back. Iterating over a query that returns a million rows can load them all into memory before the first iteration. Transaction boundaries are set by the framework's defaults, often one per request or one per statement, rather than by what the business operation needs. And the ORM writes whatever SQL its query builder produces, which may wrap an indexed column in a function and quietly defeat the index (chapter 8). None of this means avoiding ORMs. It means looking at the SQL they generate, which every ORM can log, and treating the number of queries per request as a quantity to test.

Write SQL directly where the relational model is the natural one: reports and aggregates, bulk updates (`UPDATE ... WHERE` touching a million rows should be one statement, not a million objects), window functions, and anything performance-critical you want to read in an `EXPLAIN`. Most ORMs make raw SQL easy and still map the results.

Lazy loading inside loops, especially in templates, where the query is invisible.

Joining several collections at once, multiplying rows into a cartesian product.

Loading whole objects for bulk changes instead of one set-based `UPDATE`.

Never looking at the generated SQL, so the missing index or the wrapped column goes unnoticed.

Framework-default transactions standing in for the boundaries the business logic needs.

EXERCISES

EXERCISE 4.1 Count the queries

An order list page shows 50 orders. For each it prints `order.customer.name` and `len(order.items)`, with lazy loading for both relations. How many queries does it send, and how do you get it down to a constant?

SOLUTION

One query for the orders, plus 50 for customers and 50 for items: **101 queries**, or $2N+1$. Make it constant by loading `customer` eagerly with a join (each order has exactly one customer, so the join adds columns without multiplying rows) and prefetching `items` with one `IN` query, for three queries total whatever the page size. If the page only needs the item count, as here, better still is to ask for `COUNT(*)` grouped by order in the same query and never load item objects at all.

EXERCISE 4.2 When the join is worse

An author has on average 10 books and 20 reviews. Why does eagerly loading both relations with joins in one query go badly, and what should you do instead?

SOLUTION

Joining author to books and author to reviews in one query produces every combination of a book and a review for each author: about $10 \times 20 = 200$ rows per author instead of 30, each repeating the author's columns. With a hundred authors that's 20,000 rows to send and de-duplicate in memory. Load each collection with its own query instead,

which is what prefetching does: one query for authors, one `IN` query for their books, one for their reviews. Three queries, 3,100 rows, no multiplication.

EXERCISE 4.3 Catch it in CI

Design a test that fails when someone reintroduces an N+1 query into the order list page.

SOLUTION

Count queries around the page render and assert a ceiling that doesn't depend on the data size. Most frameworks provide the counter: Django's `assertNumQueries`, SQLAlchemy event listeners on `before_cursor_execute`, Rails' notifications. The key detail is to run the test with enough rows that N+1 can't hide, say 20 orders rather than 1, because with one order, $2N+1$ and a constant 3 are the same number. Assert `queries <= 3`, so the test fails the day someone touches a lazy relation inside the loop.

CHECKLIST

- ☐ Do you know how many queries each of your main pages sends?
- ☐ Are related objects loaded eagerly or prefetched where a loop needs them?
- ☐ Do bulk changes run as single set-based statements?
- ☐ Can you see the SQL your ORM generates, and do you look?
- ☐ Is there a test that fails when the query count grows with the data?

GO DEEPER

REFERENCE Relationship loading techniques [SQLAlchemy documentation](#)

REFERENCE `select_related` and `prefetch_related` [Django documentation](#)

ESSAY Object-relational mapping is the Vietnam of computer science [Jeff Atwood](#), on [Ted Neward's essay](#)

PART II

Storage

Three chapters on where the bytes live. The fixed-size pages every table and index is cut into, and the buffer pool that decides which of them stay in memory; the B-tree, which finds one row in a billion by reading four pages; and the LSM tree, which makes writes cheap by never updating anything in place.

Pages, records and the buffer pool

3 exercises

A database doesn't read rows; it reads pages. Every table and index is a file cut into fixed-size pages, usually 4 to 16 KB, and the cost of almost any query is the number of pages it has to bring into memory. Inside each page, a slotted layout fits variable-length rows; above them, the buffer pool decides which pages stay in memory and which go back to disk.

Why pages

Storage hardware doesn't hand out single bytes. Disks and SSDs transfer blocks, the operating system caches pages, and a read of one byte costs the same as a read of the four or eight kilobytes around it. So databases organise everything in pages of a fixed size they choose: 4 KB by default in SQLite, 8 KB in PostgreSQL, 16 KB in MySQL's InnoDB. A table is a file of pages holding rows; an index is a file of pages holding keys. When a query needs a row, the database works out which page holds it, reads that page if it isn't already in memory, and looks inside.

Rows are placed in pages one of two ways. In a **heap**, PostgreSQL's default, rows go wherever there is room, in no particular order, and every index points at a row's physical location. In a **clustered** layout, InnoDB's primary key and SQLite's rowid tables, the rows themselves live in the leaves of the primary key's B-tree, sorted by key. Chapter 6 shows why that difference matters for lookups.

Inside a page

Rows vary in length, and the database needs to find any one of them quickly and move them around without breaking the indexes that point at them. The standard answer is the **slotted page**. A small header records how many slots exist and where free space ends. A **slot directory** grows forward from the front of the page, one fixed-size entry per row holding that row's offset and length. The rows themselves are packed backwards from the end. Free space is the gap in the middle, and the page is full when the two meet.



A slotted page. Indexes point at a slot, not at a byte offset, so the page can rearrange its records freely as long as it updates the slot directory. The page is full when the slots and the records meet in the middle.

EXAMPLE page.py · a working 4 KB slotted page, byte by byte

```
import struct

PAGE_SIZE = 4096
HEADER = struct.Struct("<HH") # number of slots, offset where the record area begins
SLOT = struct.Struct("<HH") # record offset, record length (length 0 means deleted)

class SlottedPage:
    """A fixed-size page. The slot directory grows from the front, records grow from the back,
    and free space is the gap between them."""
    def __init__(self):
        self.buf = bytearray(PAGE_SIZE)
        HEADER.pack_into(self.buf, 0, 0, PAGE_SIZE)
    def _slot_at(self, s):
        return HEADER.size + s * SLOT.size
    def free_space(self):
        n, start = HEADER.unpack_from(self.buf, 0)
        return start - self._slot_at(n)
    def insert(self, record):
        n, start = HEADER.unpack_from(self.buf, 0)
        if len(record) + SLOT.size > self.free_space():
            return None # full: the caller tries another
            page
        start -= len(record)
        self.buf[start:start + len(record)] = record
        SLOT.pack_into(self.buf, self._slot_at(n), start, len(record))
        HEADER.pack_into(self.buf, 0, n + 1, start)
        return n # the record id is (page, slot)
    def read(self, s):
        off, length = SLOT.unpack_from(self.buf, self._slot_at(s))
        return None if length == 0 else bytes(self.buf[off:off + length])
    def delete(self, s):
        off, _ = SLOT.unpack_from(self.buf, self._slot_at(s))
        SLOT.pack_into(self.buf, self._slot_at(s), off, 0) # the bytes stay until compaction
    def compact(self):
        n, _ = HEADER.unpack_from(self.buf, 0)
        live = [(s, self.read(s)) for s in range(n)]
        start = PAGE_SIZE
        for s, rec in live:
            if rec is not None:
                start -= len(rec)
                self.buf[start:start + len(rec)] = rec
                SLOT.pack_into(self.buf, self._slot_at(s), start, len(rec))
        HEADER.pack_into(self.buf, 0, n, start) # slots keep their numbers
```

```

page = SlottedPage()
slots = []
while (s := page.insert(f"order {len(slots):04d} ".encode().ljust(100, b"."))) is not None:
    slots.append(s)
print(f"a {PAGE_SIZE}-byte page holds {len(slots)} records of 100 bytes; {page.free_space()}
      bytes left over")
before = page.read(1)
for s in slots[::2]:
    page.delete(s)
print("deleted every other record; free space before compaction:", page.free_space())
page.compact()
print("free space after compaction:", page.free_space())
print("record in slot 1 unchanged after compaction:", page.read(1) == before, "->",
      page.read(1)[:10])

```

OUTPUT

```

a 4096-byte page holds 39 records of 100 bytes; 36 bytes left over
deleted every other record; free space before compaction: 36
free space after compaction: 2036
record in slot 1 unchanged after compaction: True -> b'order 0001'

```

Thirty-nine records of 100 bytes fill the page, with 36 bytes too few for a fortieth. Deleting every other record frees *nothing*: a delete only zeroes the slot's length, leaving the bytes where they were, because moving data on every delete would be expensive. The space comes back when the page is compacted, all at once, and 2,036 bytes reappear. The slot numbers survive compaction, so a record's id, its page and slot, stays valid. PostgreSQL calls that id the `ctid`, and every index entry stores one. A value too large for a page goes to separate overflow pages, which PostgreSQL calls TOAST, and the page keeps a pointer.

The buffer pool

The **buffer pool** is the database's own page cache: a large region of memory divided into page-sized frames. Every page access goes through it. A hit costs a memory read; a miss costs a disk read, which is roughly a thousand times slower on an SSD and a hundred thousand times slower on a spinning disk. Changed pages are marked dirty and written back later (chapter 15 explains why that's safe). When the pool is full, a page has to be evicted, and which one is the whole game. The natural choice, least recently used, has a famous weakness.

EXAMPLE `bufferpool.py` · one large scan against two eviction policies

```

import random
from collections import OrderedDict

class BufferPool:
    """Caches pages in memory. On a miss it 'reads from disk' and evicts the least recently
        used page.
    With a scan ring, large sequential scans reuse a few private buffers instead of the shared
        cache."""
    def __init__(self, capacity, scan_ring=0):
        self.capacity, self.cache, self.ring, self.ring_size = capacity, OrderedDict(),
        OrderedDict(), scan_ring
        self.disk_reads = 0

```

```

def get(self, page_id, scan=False):
    if page_id in self.cache:
        self.cache.move_to_end(page_id)
        return
    self.disk_reads += 1
    if scan and self.ring_size:
        self.ring[page_id] = True
        if len(self.ring) > self.ring_size:
            self.ring.popitem(last=False)
        return
    self.cache[page_id] = True
    if len(self.cache) > self.capacity:
        self.cache.popitem(last=False)

random.seed(5)
hot = list(range(80)) # an index and a few busy tables: the working set
for name, pool in (("plain LRU", BufferPool(100)), ("LRU + scan ring", BufferPool(100,
scan_ring=8))):
    for _ in range(2000):
        pool.get(random.choice(hot)) # warm up: the hot pages settle into the cache
    for p in range(1000, 11000):
        pool.get(p, scan=True) # someone runs a report that scans a 10,000-page
        table
    still = sum(p in pool.cache for p in hot)
    before = pool.disk_reads
    for _ in range(1000):
        pool.get(random.choice(hot))
    print(f"{name:16} hot pages still cached after the scan: {still:2}/80 | "
          f"disk reads for the next 1000 hot lookups: {pool.disk_reads - before}")

```

OUTPUT

```

plain LRU          hot pages still cached after the scan:  0/80 | disk reads for the next 1000
hot lookups: 80
LRU + scan ring    hot pages still cached after the scan: 80/80 | disk reads for the next 1000
hot lookups: 0

```

Eighty hot pages, the indexes and busy tables the application touches constantly, sit comfortably in a 100-page pool. Then one report scans a 10,000-page table. Under plain LRU, each scanned page is the most recently used, so the scan pushes every hot page out. Afterwards, the next thousand ordinary lookups need 80 disk reads to warm the cache back up. That is **sequential flooding**, and it explains the classic complaint that "the app is slow every morning after the overnight report". With a **scan ring**, large sequential scans recycle a handful of private buffers and leave the shared cache alone. PostgreSQL does exactly this for big sequential scans; InnoDB instead inserts new pages at the midpoint of its LRU list, so a page must be touched twice before it can push out the hot ones.

What this means for your schema and queries

Because pages are the unit of cost, row width matters. A table of 200-byte rows fits about forty rows in an 8 KB page; at 2,000 bytes, four. The same million-row scan reads ten times as many pages. Keep large, rarely read columns in a separate table or let the database move them out of line, select only the columns you need, and size the buffer pool so the working set, the pages your hot queries touch, fits in memory. A database whose working set fits in RAM behaves like a different machine from one whose working set doesn't.

COMMON TRAPS

Thinking in rows instead of pages when estimating what a query costs.

Wide rows, with large text or JSON stored inline, so fewer rows fit per page and every scan reads more.

Expecting deletes to shrink files. The space is reused by later inserts but returned to the operating system only by a rewrite (`VACUUM FULL` , `OPTIMIZE TABLE` , SQLite's `VACUUM`).

Running big reports on the primary, flooding the cache the application depends on.

Benchmarking with a warm cache and promising production numbers from a cold one, or the reverse.

EXERCISES

EXERCISE 5.1 Rows per page

A table has 10 million rows and 8 KB pages. Roughly how many pages, and how many gigabytes, does a full scan read if rows are 200 bytes? If they're 2,000 bytes?

SOLUTION

At 200 bytes, about 40 rows fit per page after headers and slots, so 10 million rows take about 250,000 pages, roughly 2 GB. At 2,000 bytes only 4 rows fit, so it's 2.5 million pages, about 20 GB, and the scan reads ten times as much. If the wide column is a description that most queries never read, moving it to a separate table keyed by the row's id makes the common scans ten times cheaper, and only the queries that need the description pay for the extra join.

EXERCISE 5.2 Where did the space go?

You delete half the rows of a 50 GB table, and the file is still 50 GB. Explain why, and what would make it smaller.

SOLUTION

As in the example, a delete marks space as free inside each page without moving anything; in PostgreSQL the dead rows also stay until `VACUUM` confirms no transaction can still see them (chapter 14). The free space is reused by later inserts, so the file stops growing, but it

doesn't shrink, because returning space to the operating system means moving every live row so the empty pages end up at the end of the file. Tools that rewrite the table do that: `VACUUM FULL` or `pg_repack` in PostgreSQL, `OPTIMIZE TABLE` in MySQL, `VACUUM` in SQLite. Most of them need extra space and lock or slow the table while they run. Often the right answer is to leave the space for future growth.

EXERCISE 5.3 Slow every morning

Every morning from 06:00 to about 06:20, the application's queries are three times slower than usual. A nightly export finishes at 05:58. What's happening, and what are three fixes?

SOLUTION

The export scans large tables, and the pages it reads evict the hot working set from the buffer pool and the operating system's cache. For the next twenty minutes, ordinary queries miss and go to disk until the cache warms up again. Fixes: run the export against a read replica, so the primary's cache is never touched (the usual answer); make sure the engine treats the scan as a scan, using ring buffers or midpoint insertion as in the example, and avoid scan patterns that defeat them; and pre-warm the hot pages after the export (PostgreSQL's `pg_prewarm`). Checking the buffer cache hit ratio over time would confirm the diagnosis before you change anything.

CHECKLIST

- ☐ Do you estimate query cost in pages read rather than rows returned?
- ☐ Do you know your engine's page size and whether tables are heaps or clustered?
- ☐ Are large, rarely read columns kept out of the hot table's pages?
- ☐ Does your working set fit in the buffer pool?
- ☐ Do large scans run where they can't flood the primary's cache?

GO DEEPER

COURSE CMU 15-445: storage and buffer pools Andy Pavlo

REFERENCE Database page layout PostgreSQL documentation

BOOK Database Internals Alex Petrov

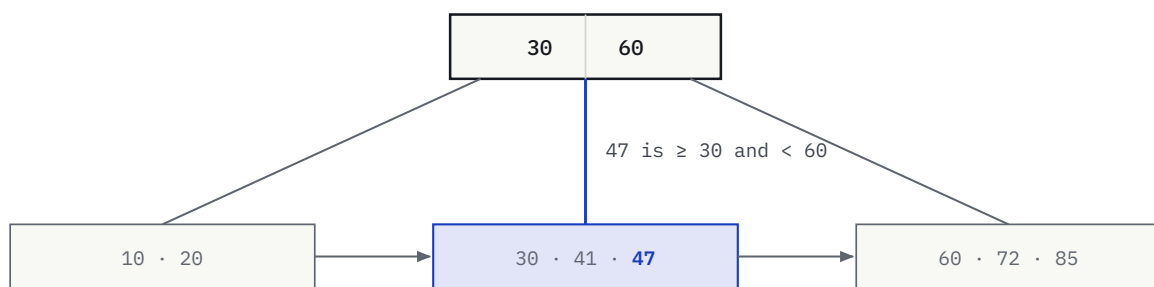
B-trees

3 exercises

The B-tree is why a database can find one row among a billion by reading four pages. It is a search tree designed for disks: each node is a whole page holding hundreds of keys, so the tree is wide and shallow. It stays balanced by splitting full nodes upwards, and its leaves are chained together so a range query reads sideways instead of climbing back up.

Wide and shallow

A binary search tree over a billion keys is about 30 levels deep, and if every level is a separate disk read, a lookup costs 30 reads. A B-tree puts as many keys in a node as fit in a page. With 8-byte keys and 8 KB pages, that's around 500 children per node, so each level multiplies the reach by 500: 500 rows at one level, 250,000 at two, 125 million at three, 60 billion at four. Better still, the root and the next level are touched by every lookup, so they stay in the buffer pool permanently. In practice a lookup in a huge table costs one or two real disk reads.



*leaves are chained: a range scan finds its first key, then walks right
a real tree has hundreds of keys per node and three or four levels*

A small B+tree. Inner nodes hold only separators that route a search; every key, and the row or pointer that goes with it, lives in a leaf. A lookup reads one node per level; a range scan reads one leaf after another along the chain.

Search, insert and split

A search starts at the root, picks the child whose range contains the key, and repeats until it reaches a leaf. An insert finds the leaf the key belongs in and adds it. If the leaf overflows, it splits in two and a separator key goes up to the parent; if the parent overflows, it splits too. When the root splits, a new root is created above it, so the tree only ever grows at the top and every leaf stays at the same depth. That is what keeps a B-tree balanced without any rebalancing pass.

EXAMPLE btree.py · a working B+tree: splits, lookups and range scans

```
import bisect, math, random

class Node:
    def __init__(self, leaf):
        self.leaf, self.keys, self.children, self.values, self.next = leaf, [], [], [], None

class BPlusTree:
    """Keys live in leaves, chained left to right; inner nodes hold only separators."""
    def __init__(self, max_keys):
        self.root, self.max_keys, self.splits = Node(leaf=True), max_keys, 0
    def search(self, key):
        node, visited = self.root, 1
        while not node.leaf:
            node = node.children[bisect.bisect_right(node.keys, key)]
            visited += 1
        i = bisect.bisect_left(node.keys, key)
        hit = i < len(node.keys) and node.keys[i] == key
        return (node.values[i] if hit else None), visited
    def insert(self, key, value):
        split = self._insert(self.root, key, value)
        if split:
            # the root split: the tree grows at the
            top
            sep, right = split
            root = Node(leaf=False)
            root.keys, root.children = [sep], [self.root, right]
            self.root = root
    def _insert(self, node, key, value):
        if node.leaf:
            i = bisect.bisect_left(node.keys, key)
            if i < len(node.keys) and node.keys[i] == key:
                node.values[i] = value
                return None
            node.keys.insert(i, key); node.values.insert(i, value)
        else:
            i = bisect.bisect_right(node.keys, key)
            split = self._insert(node.children[i], key, value)
            if not split:
                return None
            sep, right = split
            node.keys.insert(i, sep); node.children.insert(i + 1, right)
        return self._split(node) if len(node.keys) > self.max_keys else None
    def _split(self, node):
        self.splits += 1
        mid, right = len(node.keys) // 2, Node(node.leaf)
        if node.leaf:
            right.keys, right.values = node.keys[mid:], node.values[mid:]
            node.keys, node.values = node.keys[:mid], node.values[:mid]
            right.next, node.next = node.next, right
            return right.keys[0], right
            # keep the leaf chain intact
            # copy the first right key up
        sep = node.keys[mid]
        right.keys, right.children = node.keys[mid + 1:], node.children[mid + 1:]
        node.keys, node.children = node.keys[:mid], node.children[:mid + 1]
        return sep, right
            # move the middle key up
    def range(self, lo, hi):
        node = self.root
        while not node.leaf:
            node = node.children[bisect.bisect_right(node.keys, lo)]
        out = []
        while node:
            # walk the chain, no going back
            up
            for k in node.keys:
                if k > hi:
                    return out
```

```

        if k >= lo:
            out.append(k)
            node = node.next
        return out
    def height(self):
        h, node = 1, self.root
        while not node.leaf:
            node, h = node.children[0], h + 1
        return h

def levels(tree):
    rows, level = [], [tree.root]
    while level:
        rows.append(" ".join(str(n.keys) for n in level))
        level = [] if level[0].leaf else [c for n in level for c in n.children]
    return rows

small = BPlusTree(max_keys=4)
for k in range(1, 11):
    small.insert(k, f"row {k}")
print("keys 1..10, at most 4 keys per node:")
for row in levels(small):
    print(" ", row)

random.seed(1)
big = BPlusTree(max_keys=100)
keys = random.sample(range(10_000_000), 100_000)
for k in keys:
    big.insert(k, k)
value, visited = big.search(keys[12345])
print(f"\n100,000 random keys, up to 100 per node: height {big.height()}, {big.splits} splits,
      "
      f"a lookup visits {visited} nodes")
print("range 500..600000 via the leaf chain:", len(big.range(500, 600_000)), "keys, all in
      order:",
      big.range(500, 600_000) == sorted(k for k in keys if 500 <= k <= 600_000))

print("\nlevels needed with 8 KB pages about 70% full:")
for fanout, kind in ((500, "8-byte keys"), (100, "wide keys, ~80 bytes")):
    print(f" {kind:22}{fanout ~{fanout}}: " + ", ".join(
        f"{n:>13}, {rows} -> {math.ceil(math.log(n, fanout * 0.7))} levels" for n in (10**6,
        10**9)))

```

OUTPUT

```

keys 1..10, at most 4 keys per node:
[3, 5, 7]
[1, 2] [3, 4] [5, 6] [7, 8, 9, 10]

100,000 random keys, up to 100 per node: height 3, 1458 splits, a lookup visits 3 nodes
range 500..600000 via the leaf chain: 6021 keys, all in order: True

levels needed with 8 KB pages about 70% full:
8-byte keys          (fanout ~500):      1,000,000 rows -> 3 levels, 1,000,000,000 rows -> 4
levels
wide keys, ~80 bytes (fanout ~100):      1,000,000 rows -> 4 levels, 1,000,000,000 rows -> 5
levels

```

The small tree shows the shape after ten sequential inserts: a root of separators over four leaves. It also shows a real-world quirk. Because each new key was larger than everything before it, the splits always happened at the right edge, leaving half-full leaves behind and a full one at the end. Real engines detect ascending inserts and split unevenly so that pages

stay full. Random keys are worse: they land in a random leaf, so they cause splits everywhere, leave pages around 70% full, and touch a different page on every insert. That's why random UUIDs make poor clustered primary keys, and why time-ordered ids such as UUIDv7 or ULIDs exist.

The large tree holds 100,000 random keys in just three levels, with 1,458 splits along the way, and every lookup visits exactly three nodes. The range scan finds its first leaf with one descent, then walks the leaf chain, returning 6,021 keys already in order. That chain is why `ORDER BY` on an indexed column can cost nothing: the index is already sorted. The capacity lines are the arithmetic that matters. With 8-byte keys, a billion rows need four levels; wider keys mean fewer per page and an extra level.

Clustered and secondary indexes

Databases use the **B+tree** variant shown here, where inner nodes hold only separators and the data lives in the leaves. What a leaf holds depends on the index. In a **clustered index**, the leaf holds the row itself: InnoDB stores every table this way, ordered by primary key, and SQLite does the same by rowid. A **secondary index** leaf holds the indexed value plus a way to find the row. In PostgreSQL that's the row's physical location, the `ctid` from chapter 5. In InnoDB it's the primary key, so a secondary lookup is two descents, one through the secondary index and one through the clustered one. That design keeps secondary indexes valid when rows move, at the cost of the second descent.

Every index costs writes. An insert adds an entry to every index on the table, and an update to an indexed column deletes the old entry and inserts a new one. Deleted entries leave space that is reused later rather than immediately, so heavily churned indexes grow larger than their contents. The right number of indexes is the smallest number that serves your queries, which chapter 8 makes precise.

COMMON TRAPS

Random UUIDs as clustered keys, scattering every insert across the tree.

Long composite or text keys that cut the fanout and add a level to every lookup.

Indexes nobody queries, each adding write cost to every insert and update.

Forgetting the second lookup behind every secondary-index read, which a covering index avoids (chapter 8).

Assuming deletes shrink an index; heavy churn calls for an occasional rebuild.

EXERCISES

EXERCISE 6.1 How many reads?

A table has 50 million rows with an 8-byte primary key and 8 KB pages. How many levels does its B-tree have, and how many disk reads does a primary-key lookup cost with a cold cache, and with a warm one?

SOLUTION

With a fanout of about 500 at 70% fill (an effective 350), three levels reach $350^3 \approx 43$ million, just short, so it takes **four levels**, though the root will hold only a couple of keys. With a cold cache, a lookup reads one page per level: four reads. With a warm cache, the root and the second level (a few hundred pages) are permanently in memory, and often the third level too, so a lookup costs one or two disk reads: the leaf, and perhaps its parent. That's why a primary-key lookup in a table of tens of millions of rows takes well under a millisecond on an SSD.

EXERCISE 6.2 The UUID slowdown

A team switches InnoDB primary keys from auto-increment integers to random UUIDv4, and insert throughput drops sharply once the table outgrows memory. Explain why, using the example's observations.

SOLUTION

With increasing integers, every insert goes to the rightmost leaf, which is always in the buffer pool. Splits happen only at the edge, and pages fill completely. With random UUIDs, each insert lands in a random leaf. Once the table is larger than memory, that leaf usually isn't cached, so every insert costs a disk read before it can write. Splits happen all over the tree, leaving pages around 70% full, so the table and every secondary index (which store the primary key in InnoDB) grow larger and the cache holds a smaller fraction of them. Time-ordered identifiers (UUIDv7, ULID, or an integer with a separate public UUID column) keep the global uniqueness and restore the sequential insert pattern.

EXERCISE 6.3 A range of ten thousand rows

`SELECT * FROM events WHERE created_at BETWEEN :a AND :b` returns 10,000 rows using a secondary index on `created_at`. Describe the reads, and explain why the same query can be ten times faster with a different index.

SOLUTION

One descent finds the leaf holding `:a`, and the scan follows the leaf chain until it passes `:b`. With hundreds of entries per leaf, that's a few dozen leaf pages. But each entry only points at its row, so fetching the rows means 10,000 lookups into the table, scattered randomly across perhaps thousands of pages: that's where the time goes. If the query only needs a few columns, an index that includes them all (a **covering index**, chapter 8) answers it from the leaves alone. Alternatively, if `events` is clustered by `created_at`, the rows are stored in that order and the range becomes a few consecutive pages. Either way the random fetches disappear.

CHECKLIST

- ☐ Can you estimate a B-tree's height from row count, key size and page size?
- ☐ Do you know whether your tables are clustered, and by what?
- ☐ Are your primary keys sequential or time-ordered rather than random?
- ☐ Does every index earn its write cost?
- ☐ Do you know when an index read also needs a row lookup?

GO DEEPER

GUIDE Anatomy of an index Markus Winand, Use The Index, Luke

PAPER The Ubiquitous B-Tree Douglas Comer, 1979

REFERENCE The SQLite database file format SQLite documentation

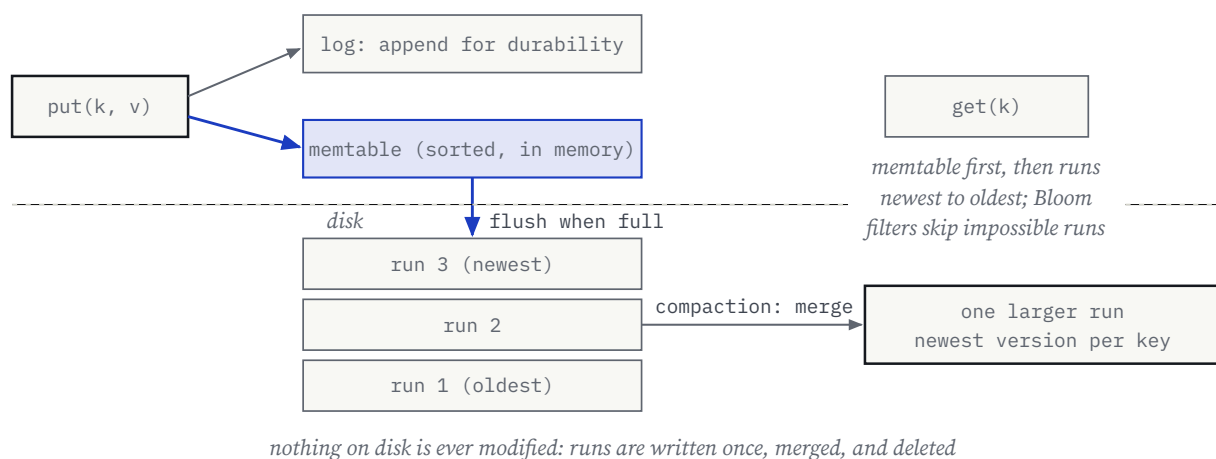
LSM trees and the price of fast writes

3 exercises

A B-tree updates data in place, which means random writes to pages all over the disk. A log-structured merge tree never updates anything: it collects writes in memory, flushes them as sorted, immutable runs, and merges the runs in the background. Writes become fast and sequential. The cost moves elsewhere, into compaction work and into reads that may have to look in several places, and every LSM design is a choice about where to pay it.

Writes go to memory first

In an LSM tree, a write is appended to a log for durability (the same idea as chapter 15) and inserted into the **memtable**, a sorted structure in memory. Nothing on disk is touched. When the memtable fills, it is written out in one sequential pass as a sorted, immutable file, usually called an **SSTable** (sorted string table), and a fresh memtable takes its place. Updates and deletes are writes too: an update is just a newer value for the key, and a delete writes a marker called a **tombstone**. Old values aren't removed; they're shadowed by newer ones.



The LSM tree. Every write is a cheap sequential append. The work of organising the data, keeping one version per key and discarding deleted keys, is deferred to compaction, which runs in the background.

Reads look everywhere a key could be

A read checks the memtable, then each run from newest to oldest, and stops at the first version it finds, which is by construction the newest. A tombstone means the key was deleted. Each run is sorted, so finding a key within one is a binary search over a small index of its pages. The problem is the number of runs: a key that was never written, or was

written long ago, means checking every run. Two tools keep this under control. **Bloom filters**, one per run, answer "definitely not here" for most runs a key isn't in, using about ten bits per key. **Compaction** merges runs so there are fewer of them to check.

EXAMPLE lsm.py · an LSM tree measured on write, read and space amplification

```
import bisect, hashlib, random

TOMBSTONE = "<deleted>"

class Bloom:
    """Answers 'definitely not here' or 'maybe here' using a few bits per key."""
    def __init__(self, keys, bits_per_key=10, k=7):
        self.m, self.k = max(64, len(keys) * bits_per_key), k
        self.bits = bytearray(self.m // 8 + 1)
        for key in keys:
            for h in self._hashes(key):
                self.bits[h // 8] |= 1 << (h % 8)
    def _hashes(self, key):
        d = hashlib.blake2b(str(key).encode(), digest_size=16).digest()
        a, b = int.from_bytes(d[:8], "little"), int.from_bytes(d[8:], "little")
        return [(a + i * b) % self.m for i in range(self.k)]
    def might_contain(self, key):
        return all(self.bits[h // 8] >> (h % 8) & 1 for h in self._hashes(key))

class SSTable:
    """A sorted run written once and never modified."""
    def __init__(self, items, bloom):
        self.keys, self.vals = [k for k, _ in items], [v for _, v in items]
        self.bloom = Bloom(self.keys) if bloom else None
    def find(self, key):
        i = bisect.bisect_left(self.keys, key)
        return (True, self.vals[i]) if i < len(self.keys) and self.keys[i] == key else (False, None)

class LSMTree:
    def __init__(self, memtable_size=100, compact_at=None, bloom=False):
        self.mem, self.runs = {}, [] # runs: newest first
        self.memtable_size, self.compact_at, self.bloom = memtable_size, compact_at, bloom
        self.user_writes = self.disk_writes = self.probes = 0
    def put(self, key, value):
        self.user_writes += 1
        self.mem[key] = value # writes only ever touch memory...
        if len(self.mem) >= self.memtable_size:
            self._flush()
    def delete(self, key):
        self.put(key, TOMBSTONE) # ...even deletes, which are writes
    def _flush(self):
        items = sorted(self.mem.items())
        self.mem = {}
        self.runs.insert(0, SSTable(items, self.bloom)) # ...then go to disk as one sorted
        run
        self.disk_writes += len(items)
        if self.compact_at and len(self.runs) >= self.compact_at:
            self._compact()
    def _compact(self):
        merged = {}
        for run in reversed(self.runs): # oldest first, so newer values win
            merged.update(zip(run.keys, run.vals))
        items = [(k, v) for k, v in sorted(merged.items()) if v != TOMBSTONE]
        self.runs = [SSTable(items, self.bloom)] # rewrite everything as one run
        self.disk_writes += len(items)
    def get(self, key):
```

```

    if key in self.mem:
        return None if self.mem[key] == TOMBSTONE else self.mem[key]
    for run in self.runs:
        if run.bloom and not run.bloom.might_contain(key):
            continue # skipped without reading it
        self.probes += 1
        found, value = run.find(key)
        if found:
            return None if value == TOMBSTONE else value
    return None
def stored(self):
    return sum(len(r.keys) for r in self.runs) + len(self.mem)

random.seed(7)
writes = [(random.randrange(5000), i) for i in range(20_000)] # 5,000 keys, each updated
~4 times
present = random.sample(range(5000), 1000)
missing = [10**6 + i for i in range(1000)]
print(f"{'':28}{ 'runs':>5}{ 'write amp':>11}{ 'space amp':>11}{ 'reads/hit':>11}{ 'reads/miss':>12}")
for name, tree in (("no compaction, no Bloom", LSMTree()),
                   ("no compaction, Bloom", LSMTree(bloom=True)),
                   ("compact at 4 runs, Bloom", LSMTree(compact_at=4, bloom=True))):
    for k, v in writes:
        tree.put(k, v)
    live = len({k for k, _ in writes})
    tree.probes = 0
    for k in present:
        tree.get(k)
    per_hit = tree.probes / len(present)
    tree.probes = 0
    for k in missing:
        tree.get(k)
    per_miss = tree.probes / len(missing)
    print(f"{name:28}{len(tree.runs):5}{tree.disk_writes / tree.user_writes:10.1f}x"
          f"{tree.stored() / live:10.1f}x{per_hit:11.1f}{per_miss:12.1f}")

tree.delete(present[0])
print("\nafter delete:", tree.get(present[0]), "| the tombstone hides the old value until
      compaction removes both")

```

OUTPUT

	runs	write amp	space amp	reads/hit	reads/miss
no compaction, no Bloom	198	1.0x	4.0x	48.4	198.0
no compaction, Bloom	198	1.0x	4.0x	1.4	2.0
compact at 4 runs, Bloom	3	13.4x	1.0x	1.0	0.0

after delete: None | the tombstone hides the old value until compaction removes both

The table holds the whole trade-off in three rows. Without compaction, writes are as cheap as they can be: each entry is written to disk exactly once, a write amplification of 1.0. But all 20,000 versions of 5,000 keys stay on disk, a space amplification of 4.0, and reads are dreadful: finding a present key checks 48 runs on average, and proving a key absent checks all 198. Bloom filters rescue the reads without changing anything else, down to 1.4 and 2.0 runs. Compaction rescues the space too, down to 1.0, with about one run read per lookup, but it pays by rewriting live data again and again: every entry was written 13.4 times. This is the **RUM conjecture** in miniature: you can optimise for reads, updates or memory (space), and improving one costs at least one of the others.

Compaction strategies

Real systems compact more cleverly than "merge everything at four runs", but they choose between the same costs. **Size-tiered** compaction, Cassandra's long-time default, merges runs of similar size: it rewrites data fewer times, but may hold several versions of a key for a while, and needs free disk space as large as the runs being merged. **Leveled** compaction, used by LevelDB and RocksDB, keeps data in levels that each grow about ten times larger, with no overlapping keys within a level. Space and read costs stay tightly bounded, and data gets rewritten more often. Tombstones are their own hazard. A tombstone can only be dropped once compaction reaches the oldest run that might hold the key, so a workload that deletes heavily can leave reads wading through tombstones in the meantime.

B-tree or LSM tree?

B-trees suit read-heavy workloads that want predictable latency: every read is a fixed number of page reads, and no background process competes for the disk. LSM trees suit write-heavy workloads: event logs, metrics, messaging, anything that ingests far more than it updates. They also compress well, because runs are immutable and sorted, and they write sequentially. They power RocksDB (and through it MyRocks, CockroachDB's Pebble and TiKV), Cassandra, ScyllaDB and most time-series stores. Discord stores its message history this way: it moved it to Cassandra in 2017 and to ScyllaDB in 2023. B-trees have write amplification too, which is easy to forget: changing 100 bytes rewrites an 8 KB page, and if the change touches five indexes, five pages.

COMMON TRAPS

Benchmarking before compaction reaches a steady state, measuring cheap writes and ignoring the debt they create.

Running out of disk during compaction, which writes the new run before deleting the old ones.

Delete-heavy workloads that leave reads scanning past piles of tombstones.

Long range scans across many runs, which can't use Bloom filters and must merge every run.

Write stalls: when compaction falls behind, LSM engines deliberately slow or stop writes.

EXERCISES

EXERCISE 7.1 The worst lookup

Why is looking up a key that doesn't exist the worst case for an LSM tree, and what exactly do Bloom filters change? Use the example's numbers.

SOLUTION

A read stops at the first run that holds the key, so a present key, especially a recently written one, is often found early. An absent key can only be proven absent by checking every run: 198 of them without compaction. A Bloom filter for each run answers "definitely not here" for almost all of those runs without reading them, with a false-positive rate of about 1% at ten bits per key, so the cost dropped from 198 run reads to 2.0. Filters can't help a key that is present (it must be read somewhere) or a range scan (a filter answers "is this exact key here?", not "is anything in this range here?").

EXERCISE 7.2 Full disk, stopped writes

A Cassandra node with a disk at 70% full stops accepting writes during a large compaction. Explain how a disk at 70% can run out of space.

SOLUTION

Compaction writes the merged run before it can delete the runs it merged. With size-tiered compaction, merging the four largest runs, which may together be most of the node's data, needs free space roughly equal to their total size while both copies exist. At 70% full, a compaction over runs totalling more than the remaining 30% runs out of disk midway; the node can neither finish compaction nor accept more writes. That's why size-tiered deployments keep up to half the disk free, and why leveled compaction, which merges small pieces at a time, needs far less headroom.

EXERCISE 7.3 Choose the structure

Choose a B-tree engine or an LSM engine for each, and say why: (a) sensor readings at 200,000 writes a second, queried by device and recent time range; (b) a bank ledger read ten times more often than it's written, with strict latency targets; (c) a chat application's message history.

SOLUTION

- a. **LSM.** Ingest dominates, writes arrive roughly in time order (which compacts cheaply), old data is rarely updated, and immutable sorted runs compress repetitive readings well.

- b. **B-tree.** Reads dominate and must be predictable. A B-tree reads a fixed number of pages and has no compaction competing for disk at unpredictable moments. A relational engine also gives the ledger the transactions and constraints it needs.
- c. **Usually LSM.** Messages are written constantly, almost never edited, and read in recent-first ranges per channel, which is why Discord runs on Cassandra and ScyllaDB. A B-tree engine also works at moderate scale, and many products start there; the switch becomes worthwhile when write volume makes page-level random writes the bottleneck.

CHECKLIST

- ☐ Can you trace a write and a read through memtable, runs and compaction?
- ☐ Can you name the three amplifications and what trades between them?
- ☐ Do you know which compaction strategy your engine uses and its disk headroom needs?
- ☐ Do you understand what Bloom filters can and can't speed up?
- ☐ Would you choose an LSM or a B-tree engine for your workload, and why?

GO DEEPER

PAPER The Log-Structured Merge-Tree O'Neil, Cheng, Gawlick & O'Neil, 1996

REFERENCE Compaction RocksDB wiki

CASE STUDY How Discord stores trillions of messages Discord engineering

PART III

Queries

Four chapters on how SQL becomes work. The rules that decide whether an index can be used at all, the operators and join algorithms a plan is built from, the planner's estimates and how they go wrong, and a real tuning session that takes a million-row report from 68 milliseconds to about 3.

Indexes the planner will use

3 exercises

An index only helps a query that can use it, and whether it can is decided by a few mechanical rules: the column must appear bare, not inside a function; the index's leading columns must be constrained; and equality columns must come before range columns. Learn the rules, check every important query's plan, and keep the fewest indexes that serve your real queries.

Reading the plan

Every database will tell you how it intends to run a query. SQLite's `EXPLAIN QUERY PLAN` prints one line per step; PostgreSQL and MySQL's `EXPLAIN` print a tree, and their `EXPLAIN ANALYZE` actually runs the query and adds real row counts and timings to each step. The words to look for are few. A **scan** reads a whole table or index (`SCAN` in SQLite, `Seq Scan` in PostgreSQL). A **search** descends an index to the matching entries (`SEARCH ... USING INDEX`, `Index Scan`). A **covering** or **index-only** scan answers from the index without touching the table at all. The example below runs four index lessons against SQLite's real planner, on a 200,000-row orders table, and measures the work as the number of instructions SQLite's engine executes, a count that comes out identical on every run.

EXAMPLE `indexes.py` · four rules, checked against the planner

```
import random, sqlite3
from datetime import date, timedelta

random.seed(8)
db = sqlite3.connect(":memory:")
db.execute("""CREATE TABLE orders(id INTEGER PRIMARY KEY, customer_id INTEGER, status TEXT,
                                   created_at TEXT, total_cents INTEGER)""")

start = date(2025, 1, 1)
db.executemany("INSERT INTO orders(customer_id, status, created_at, total_cents) VALUES (?, ?,
?, ?)",
              ((random.randint(1, 20_000),
                 random.choices(["shipped", "pending", "cancelled"], [85, 10, 5])[0],
                 (start + timedelta(days=random.randrange(638))).isoformat(),
                 random.randint(500, 50_000)) for _ in range(200_000)))

def plan(sql):
    return " / ".join(row[-1] for row in db.execute("EXPLAIN QUERY PLAN " + sql))

def work(sql):
    """Virtual-machine steps SQLite executes for the query: a deterministic measure of
    effort."""
    steps = 0
    def tick():
        nonlocal steps
        steps += 1
        return 0
    db.set_progress_handler(tick, 10)  # called every 10 instructions
```

```

db.execute(sql).fetchall()
db.set_progress_handler(None, 0)
return steps * 10

def show(label, sql, count=True):
    effort = f"{work(sql):>10,} steps" if count else " " * 16
    print(f" {label:34} {effort} {plan(sql)}")

def reset(*indexes):
    for (name,) in db.execute("SELECT name FROM sqlite_master WHERE type =
        'index'").fetchall():
        db.execute(f"DROP INDEX {name}")
    for ddl in indexes:
        db.execute(ddl)

q1 = "SELECT * FROM orders WHERE customer_id = 4242"
print("1. equality lookup")
reset(); show("no index", q1)
reset("CREATE INDEX by_customer ON orders(customer_id)"); show("index on customer_id", q1)

q2 = "SELECT id FROM orders WHERE status = 'pending' AND created_at >= '2026-09-01'"
print("2. composite index: column order")
reset("CREATE INDEX by_created_status ON orders(created_at, status)"); show("(created_at,
    status): range first", q2)
reset("CREATE INDEX by_status_created ON orders(status, created_at)"); show("(status,
    created_at): equality first", q2)

print("3. a function on the column hides it from the index")
reset("CREATE INDEX by_created ON orders(created_at)")
show("substr(created_at, 1, 7) = ...", "SELECT id FROM orders WHERE substr(created_at, 1, 7) =
    '2026-09'")
show("created_at range, same rows", "SELECT id FROM orders WHERE created_at >= '2026-09-01' AND
    created_at < '2026-10-01'")

q4 = "SELECT created_at, total_cents FROM orders WHERE customer_id = 4242"
print("4. covering index: answer from the index alone (each table lookup is one step here, so
    see chapter 11 for timings)")
reset("CREATE INDEX by_customer ON orders(customer_id)"); show("index on customer_id", q4,
    count=False)
reset("CREATE INDEX by_customer_cover ON orders(customer_id, created_at, total_cents)");
    show("index includes every column used", q4, count=False)

```

OUTPUT

```

1. equality lookup
  no index                                600,070 steps  SCAN orders
  index on customer_id                    120 steps   SEARCH orders USING INDEX by_customer
    (customer_id=?)

2. composite index: column order
  (created_at, status): range first      29,800 steps  SEARCH orders USING COVERING INDEX
    by_created_status (created_at>?)
  (status, created_at): equality first    3,730 steps  SEARCH orders USING COVERING INDEX
    by_status_created (status=? AND created_at>?)

3. a function on the column hides it from the index
  substr(created_at, 1, 7) = ...          818,630 steps  SCAN orders USING COVERING INDEX
    by_created
  created_at range, same rows             37,250 steps  SEARCH orders USING COVERING INDEX
    by_created (created_at>? AND created_at<?)

4. covering index: answer from the index alone (each table lookup is one step here, so see
  chapter 11 for timings)
  index on customer_id                    SEARCH orders USING INDEX by_customer
    (customer_id=?)
  index includes every column used        SEARCH orders USING COVERING INDEX
    by_customer_cover (customer_id=?)

```

Rule 1: an index turns a scan into a search

Without an index, finding one customer's orders means reading all 200,000 rows: 600,070 steps. With one, it's a descent and a few entries: 120 steps, five thousand times less work. That ratio grows with the table, because a scan is linear and a B-tree descent is logarithmic.

Rule 2: equality columns first, then range

A composite index is sorted by its first column, then by its second within equal first values, like a phone book sorted by surname, then first name. That ordering decides what the index can find. With `(status, created_at)`, all the `'pending'` entries sit together, sorted by date, so "pending since September" is one contiguous slice: 3,730 steps. With `(created_at, status)`, the index can only use the date range, and must read every entry from September 1st onwards, of every status, checking each one: 29,800 steps for the same answer. The general rule: put columns compared with `=` first and the one range column last. An index on `(a, b, c)` serves queries on `a`, on `a` and `b`, and on all three, but not on `b` alone. This is the **leftmost prefix** rule.

index on (status, created_at)

cancelled · 2026-09-14
pending · 2026-08-30
pending · 2026-09-01
pending · 2026-09-02
pending · 2026-09-05
shipped · 2025-01-01
shipped · 2025-01-02

one contiguous slice: read it and stop
3,730 steps

index on (created_at, status)

2026-08-31 · shipped
2026-09-01 · cancelled
2026-09-01 · pending
2026-09-01 · shipped
2026-09-02 · pending
2026-09-02 · shipped
2026-09-03 · shipped ...

matches scattered: read everything after Sept 1
29,800 steps

Why column order matters. The query is `status = 'pending' AND created_at >= '2026-09-01'`. With the equality column first, the matches are adjacent in the index. With the range column first, they're interleaved with every other status, so the index narrows the search to a date range and no further.

Rule 3: keep the column bare

An index on `created_at` is sorted by `created_at`, not by `substr(created_at, 1, 7)`. Wrap the column in a function and the planner has no ordering to search, so it falls back to reading every entry and computing the function for each: 818,630 steps. The same rows, asked for as a range on the bare column, cost 37,250. This catches out `WHERE date(created_at) = ...`, `WHERE lower(email) = ...`, `WHERE total_cents / 100 > 50`, and

implicit conversions, such as comparing a text column to a number. Rewrite the condition so the bare column is compared to a value, or, when you genuinely query by the expression, index the expression itself: `CREATE INDEX ON users(lower(email))` .

Rule 4: a covering index never touches the table

An ordinary index finds the matching entries and then fetches each row from the table to get the other columns: one more lookup per row, often to a random page. If the index contains every column the query uses, the plan says `COVERING INDEX` (PostgreSQL: `Index Only Scan`) and the table is never read. The instruction count can't show the saving, because each table lookup is a single instruction however many pages it reads, so chapter 11 measures it with a stopwatch instead. On a million rows, making the index covering cut the report from 31.3 ms to 3.1. PostgreSQL lets you add columns to an index purely for covering, without sorting by them: `CREATE INDEX ON orders(customer_id) INCLUDE (total_cents)` .

Selectivity, and what indexes cost

An index pays off when it narrows the rows to a small fraction. An index on `status` , where 85% of rows are `'shipped'` , narrows almost nothing, and following it to 170,000 scattered rows is slower than reading the table in order: chapter 11 shows exactly that. If a small subset is what you query, a **partial index** indexes just that subset: `CREATE INDEX ON orders(created_at) WHERE status = 'pending'` . And every index has a price: space, cache, and work on every insert and update. Find the unused ones (PostgreSQL's `pg_stat_user_indexes` counts scans per index) and the redundant ones, since `(a)` is already served by `(a, b)` , and drop them.

COMMON TRAPS

One index per column, instead of composite indexes shaped like your queries.

Range column first in a composite index, so equality conditions after it can't narrow the search.

Functions and implicit casts on the indexed column, silently turning a search into a scan.

Leading wildcards such as `LIKE '%son'` , which no ordinary B-tree index can serve.

Low-selectivity indexes that the planner may use and regret.

EXERCISES

EXERCISE 8.1 Design the index

A multi-tenant app's busiest query is `WHERE tenant_id = ? AND created_at > ? ORDER BY created_at LIMIT 50`. Which index serves it best, and why not `(created_at, tenant_id)`?

SOLUTION

`(tenant_id, created_at)`. The equality column comes first, so one tenant's entries are contiguous and sorted by date; the range then selects a slice of them, already in the order the query wants, so `ORDER BY ... LIMIT 50` reads 50 entries and stops, with no sort. With `(created_at, tenant_id)`, the index can only use the date range, so it reads every tenant's rows after that date, checking each one's tenant. For a big tenant that may work out fine; for a small tenant in a busy system it reads thousands of entries to find fifty.

EXERCISE 8.2 Make it sargable

Rewrite each condition so an ordinary index on the column can be used: (a) `date(created_at) = '2026-09-14'`; (b) `total_cents / 100 > 50`; (c) `lower(email) = 'ana@example.com'`.

SOLUTION

- `created_at >= '2026-09-14' AND created_at < '2026-09-15'`: a half-open range on the bare column.
- `total_cents > 5000`: move the arithmetic to the constant side.
- This one can't be rewritten on the bare column, because case-insensitive matching genuinely depends on the expression. Either store a normalised copy (lower-case the email on write) or create an expression index, `CREATE INDEX ON users(lower(email))`, which the planner uses when the query's expression matches the index's exactly.

EXERCISE 8.3 Prune the indexes

A table has indexes on `(a)`, `(a, b)`, `(b)` and `(a, b, c)`. The queries are `WHERE a = ?`, `WHERE a = ? AND b = ?`, `WHERE b = ?`, and `WHERE a = ? AND b = ? AND c > ?`. Which indexes can you drop?

SOLUTION

Drop `(a)` and `(a, b)`. By the leftmost prefix rule, `(a, b, c)` serves `a = ?`, `a = ? AND b = ?` and the full query, so both shorter indexes are redundant. Keep `(b)`, because no index starting with `a` can serve a query constraining only `b`. That's two indexes instead of four, which halves the index maintenance on every write. The one caveat: a narrower index is slightly smaller and faster to scan, which matters only if a query reads a large share of it.

CHECKLIST

- ☐ Have you read the plan of every important query?
- ☐ Do composite indexes put equality columns before the range column?
- ☐ Are indexed columns compared bare, without functions or casts?
- ☐ Do your hottest reads use covering indexes where it's worth it?
- ☐ Have you dropped unused and redundant indexes?

GO DEEPER

BOOK [Use The Index](#), Luke Markus Winand

REFERENCE [Query planning SQLite documentation](#)

REFERENCE [Using EXPLAIN PostgreSQL documentation](#)

How a query executes: operators and joins

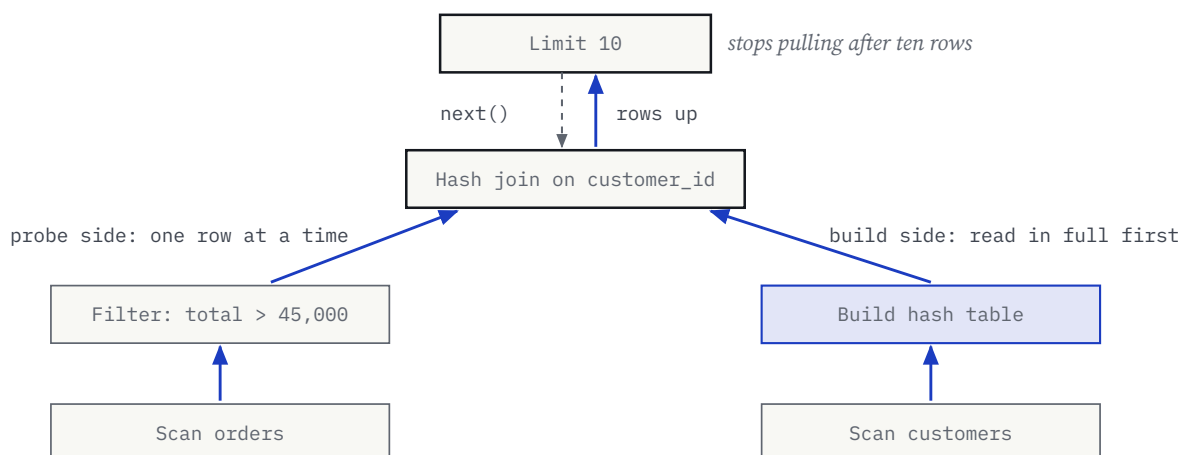
3 exercises

A plan is a tree of operators, each doing one simple job: scan, filter, join, sort, aggregate. Rows flow up the tree one at a time, so a query can start returning results before it has read everything, unless some operator has to see all of its input first. And for every join, the algorithm the planner picks is often the biggest single factor in what the query costs.

The iterator model

Most databases execute a plan with the **iterator** or **Volcano** model, named after the research system that formalised it. Every operator has the same three methods: `open`, `next` and `close`. Calling `next()` on the top operator makes it call `next()` on its children until it has a row to return. A filter pulls rows from the scan below until one passes; a projection pulls one row and trims its columns. Rows stream upwards, one at a time, through the whole tree, without the intermediate results ever being stored. This is **pipelining**, and it's why `LIMIT 10` can return after reading a few dozen rows of a huge table.

Some operators can't pipeline. A sort can't return its first row until it has seen every row, because the last one might sort first. A hash join must read its whole build side before it can probe it, and a grouping aggregate must see every row of a group. These **blocking** operators break the stream: everything below them runs to completion first. Spotting them in a plan explains most "why didn't LIMIT help?" mysteries.



A plan as an operator tree. Each `next()` call pulls one row up through the pipeline. The hash join's build side is blocking: it reads all its input before the first probe. Its probe side streams, which is what lets `Limit` stop the whole tree early.

Four ways to join

A join combines rows from two inputs where a condition holds, and there are four basic algorithms. A **nested loop** compares every row of one input with every row of the other: simple, and fine when one side has a handful of rows. An **index nested loop** looks each outer row's key up in an index on the inner side; it's the workhorse of transactional queries, where the outer side is small and the inner side is indexed. A **hash join** builds a hash table on the smaller input and streams the larger one past it; it suits large inputs with no useful index, works only for equality conditions, and needs memory for the table. A **sort-merge join** sorts both inputs on the key and walks them together once; it's cheapest when both inputs already arrive sorted, from an index for instance, and it also handles range conditions.

EXAMPLE joins.py · four join algorithms on the same data, and the iterator model

```
import bisect, functools, random

random.seed(3)
customers = [(cid, random.choice(["EU", "US", "APAC"])) for cid in range(1, 1001)]
orders = [(oid, random.randint(1, 1000), random.randint(100, 50_000)) for oid in range(1, 20_001)]

def nested_loop(outer, inner):
    """For every outer row, scan every inner row."""
    out, work = [], 0
    for o in outer:
        for c in inner:
            work += 1
            if o[1] == c[0]:
                out.append(o + c)
    return out, work

def index_nested_loop(outer, inner):
    """For every outer row, look the key up in a sorted index (a stand-in for a B-tree)."""
    keys = sorted(c[0] for c in inner)
    by_key = {c[0]: c for c in inner}
    out, work = [], 0
    for o in outer:
        lo, hi = 0, len(keys)
        while lo < hi:
            work += 1
            mid = (lo + hi) // 2
            if keys[mid] < o[1]: lo = mid + 1
            else: hi = mid
        if lo < len(keys) and keys[lo] == o[1]:
            out.append(o + by_key[o[1]])
    return out, work

def hash_join(probe, build):
    """Build a hash table on the smaller input, then stream the larger one past it."""
    table, work = {}, 0
    for c in build:
        work += 1
        table.setdefault(c[0], []).append(c)
    out = []
    for o in probe:
        work += 1
        for c in table.get(o[1], ()):
            out.append(o + c)
    return out, work
```

```

def merge_join(left, right):
    """Sort both inputs on the key, then walk them together once."""
    comparisons = 0
    def counted(key):
        def cmp(a, b):
            nonlocal comparisons
            comparisons += 1
            return (key(a) > key(b)) - (key(a) < key(b))
        return functools.cmp_to_key(cmp)
    L = sorted(left, key=counted(lambda o: o[1]))
    R = sorted(right, key=counted(lambda c: c[0]))
    out, i, j, steps = [], 0, 0, 0
    while i < len(L) and j < len(R):
        steps += 1
        if L[i][1] < R[j][0]: i += 1
        elif L[i][1] > R[j][0]: j += 1
        else:
            out.append(L[i] + R[j]); i += 1
            # customer ids are unique on the
            # right
    return out, comparisons + steps

results = {}
for name, fn in (("nested loop", nested_loop), ("index nested loop", index_nested_loop),
                ("hash join", hash_join), ("sort-merge join", merge_join)):
    rows, work = fn(orders, customers)
    results[name] = sorted(rows)
    print(f'{name:18} {len(rows):6} rows {work:>12} units of work')
print("all four produce the same rows:", len({tuple(map(tuple, r)) for r in results.values()})
      == 1)

# The iterator model: each operator pulls rows from the one below, one at a time.
def scan(rows, stats):
    for r in rows:
        stats["read"] += 1
        yield r
def where(rows, predicate):
    return (r for r in rows if predicate(r))
def limit(rows, n):
    for i, r in enumerate(rows):
        if i == n:
            return
        yield r

stats = {"read": 0}
top = list(limit(where(scan(orders, stats), lambda o: o[2] > 45_000), 10))
print(f"\nLIMIT 10 over a filter: {len(top)} rows returned after reading {stats['read']} of
      {len(orders)}")
stats = {"read": 0}
top = list(limit(sorted(where(scan(orders, stats), lambda o: o[2] > 45_000), key=lambda o: -
                        o[2]), 10))
print(f"add ORDER BY (a blocking sort): {len(top)} rows returned after reading {stats['read']}
      of {len(orders)}")

```

OUTPUT

```

nested loop          20000 rows      20,000,000 units of work
index nested loop    20000 rows      199,578 units of work
hash join            20000 rows       21,000 units of work
sort-merge join      20000 rows      277,244 units of work
all four produce the same rows: True

```

```

LIMIT 10 over a filter: 10 rows returned after reading 77 of 20000
add ORDER BY (a blocking sort): 10 rows returned after reading 20000 of 20000

```

All four produce the same 20,000 rows; the work differs by a factor of a thousand. The plain nested loop does 20 million comparisons, one for every pair. The index nested loop needs about ten binary-search steps per order, 199,578 in all. The sort-merge join spends most of its 277,244 units sorting, and would be nearly free if both inputs arrived sorted. The hash join wins at 21,000: one pass to build a table of 1,000 customers, one pass to stream 20,000 orders past it. The last two lines show the iterator model at work. `LIMIT 10` over a filter read just 77 rows before it had its ten and stopped pulling. Add `ORDER BY`, and the sort, a blocking operator, read all 20,000 first.

Which join, and who decides

The planner chooses the algorithm for each join, along with the order in which tables are joined, from its estimates of how many rows each side will produce (chapter 10). Engines differ in what they offer. PostgreSQL and SQL Server implement all of them. MySQL relied on nested loops until it added hash joins in version 8.0.18. SQLite implements only nested loops, and when no index fits, it builds a temporary one just for the query, which its plans report as an automatic index. In PostgreSQL plans, look for `Nested Loop`, `Hash Join` with a `Hash` node beneath it, and `Merge Join`, usually above a `Sort`. A hash join whose build side exceeds its memory budget (`work_mem`) spills to disk in batches and slows sharply; `EXPLAIN ANALYZE` reports the batch count.

COMMON TRAPS

Assuming tables are joined in the order you wrote them. The planner reorders joins freely.

No index on a foreign key column, so every join from the parent becomes a scan of the child.

`ORDER BY ... LIMIT` without a supporting index, which forces a full blocking sort.

Joining on expressions or mismatched types, which hides the indexes on both sides.

Hash joins spilling to disk because the memory budget is smaller than the build side.

EXERCISES

EXERCISE 9.1 Pick the algorithm

Which join algorithm fits each case? (a) one customer's 5 orders joined to `products`, which has a primary-key index; (b) a nightly report joining two 50-million-row tables on `customer_id`, neither sorted; (c) two inputs that both arrive sorted by date from index scans, joined on date.

SOLUTION

- a. **Index nested loop.** Five index lookups beat building a hash table over the whole products table by orders of magnitude.
- b. **Hash join,** building on the smaller input (or on the filtered one, if the report filters first). It reads each table once. Check the memory budget, because a 50-million-row build side may spill; sort-merge is the alternative if both sides can be sorted cheaply.
- c. **Sort-merge join.** The sorting is already done by the indexes, so the join is a single synchronised walk through both inputs, which also preserves the date order for any `ORDER BY` above it.

EXERCISE 9.2 The slow LIMIT

`SELECT * FROM orders WHERE total > 45000 LIMIT 10` returns instantly. Adding `ORDER BY total DESC` makes it take four seconds on a large table. Explain, and fix it.

SOLUTION

Without `ORDER BY`, the plan pipelines: the scan streams rows through the filter and `LIMIT` stops after ten, as the example's 77 rows showed. `ORDER BY total DESC` inserts a blocking sort, which must read every matching row before it can know the top ten. Fix it with an index on `total`. The index already holds rows in `total` order, so the planner reads it backwards from the highest value, filters as it goes, and stops after ten: the sort disappears from the plan and the query pipelines again.

EXERCISE 9.3 Count the work

Estimate the comparisons for a nested loop join of two 10,000-row tables, then for an index nested loop where the inner side has a B-tree index on the join column.

SOLUTION

A plain nested loop compares every pair: $10,000 \times 10,000 = 100 \text{ million}$ comparisons. An index nested loop does one index lookup per outer row, about $\log_2(10,000) \approx 14$ steps in a binary search, or 2 to 3 page reads in a real B-tree, so roughly $10,000 \times 14 = 140,000$ steps, about 700 times less. That's the same ratio the example shows between its nested

loop and its index nested loop, and why a missing index on a join column is the most common cause of a query that "suddenly" got slow as a table grew.

CHECKLIST

- ☐ Can you read a plan as a tree and say which operators are blocking?
- ☐ Do you know the four join algorithms and when each wins?
- ☐ Are foreign key columns indexed?
- ☐ Do `ORDER BY ... LIMIT` queries have an index that delivers the order?
- ☐ Do you know your engine's memory budget for sorts and hash joins?

GO DEEPER

COURSE CMU 15-445: query execution and join algorithms Andy Pavlo

REFERENCE The next-generation query planner SQLite documentation

REFERENCE Planner and optimizer PostgreSQL documentation

The planner and its statistics

3 exercises

The planner chooses between plans by estimating how many rows each step will produce and what each step will cost. It can't run the query to find out, so it estimates from statistics: row counts, distinct values, the most common values, histograms. When a plan is bad, the estimate behind it is almost always wrong, and the fix is usually better statistics, not a hint that overrules the planner.

Cost-based optimisation

For any non-trivial query there are many correct plans: which index to use for each table, which order to join the tables in, which algorithm for each join. A **cost-based optimiser**, an idea that dates back to IBM's System R in the 1970s, enumerates candidate plans, estimates each one's cost with a model that charges for page reads and CPU work, and picks the cheapest. The search space grows explosively with the number of tables, so planners use dynamic programming for small joins and heuristics beyond that; PostgreSQL switches to a genetic search above twelve tables. All of this rests on one input that's easy to forget: the estimated number of rows flowing out of each step. Get that wrong and the cheapest-looking plan can be the slowest.

What the planner knows

Statistics are gathered by `ANALYZE`, run manually or by a background process (PostgreSQL's autovacuum, MySQL's persistent statistics). For each table they record the row count. For each column, typically: the fraction of NULLs, the number of distinct values, a list of the **most common values** with their frequencies, and a **histogram** of the rest. To estimate `WHERE country = 'US'`, the planner checks the common-values list; to estimate a range, it reads the histogram. For combinations of conditions, it usually assumes they're **independent** and multiplies their selectivities. Each of those steps can go wrong.

EXAMPLE `planner.py` · how estimates go wrong, and what a wrong estimate does

```
import random, sqlite3
from collections import Counter

random.seed(10)
N = 100_000
countries = ["US"] * 70 + [f"C{i:02d}" for i in range(30)] # 70% US, then a
long tail
rows = []
for _ in range(N):
    country = random.choice(countries)
    city = random.choice(["Paris", "Lyon", "Nice"]) if country == "C07" else f"city-
{random.randrange(400)}"
    rows.append((country, city))
```

```

# What a planner keeps per column: row count, distinct values, and the most common values.
distinct = {"country": len({c for c, _ in rows}), "city": len({t for _, t in rows})}
mcv = dict(Counter(c for c, _ in rows).most_common(5))

def actual(pred):
    return sum(1 for r in rows if pred(r))

print("1. skew: WHERE country = 'US'")
print(f"    uniform guess N/distinct = {N // distinct['country']}>6,}    with most-common-values
      list = "
      f"{mcv['US']}>6,}    actual = {actual(lambda r: r[0] == 'US'),}")

print("2. correlation: WHERE country = 'C07' AND city = 'Paris'")
sel_country = actual(lambda r: r[0] == "C07") / N
sel_city = actual(lambda r: r[1] == "Paris") / N
estimate = round(N * sel_country * sel_city)
true_rows = actual(lambda r: r[0] == "C07" and r[1] == "Paris")
print(f"    independence guess = {estimate}>7,}    actual = {true_rows:,}    (every Paris row is
      in C07)")

print("3. the plan follows the estimate: join those rows to a 600-page customers table")
def cost_nested_loop(outer_rows): return outer_rows * 4          # one index descent, ~4 pages,
per row
def cost_hash_join(outer_rows): return 600 + outer_rows * 0.1    # read customers once, probe
in memory
for label, rows_in in (("estimated", estimate), ("actual", true_rows)):
    nl, hj = cost_nested_loop(rows_in), cost_hash_join(rows_in)
    print(f"    {label:9} {rows_in:>5} outer rows: nested loop {nl:>7,.0f} hash join
          {hj:>7,.0f} -> "
          f"{'nested loop' if nl < hj else 'hash join'}")

print("4. a real planner, before and after it gathers statistics")
db = sqlite3.connect(":memory:")
db.execute("CREATE TABLE orders(id INTEGER PRIMARY KEY, status TEXT, created_at TEXT)")
db.executemany("INSERT INTO orders(status, created_at) VALUES (?, ?)",
               ((random.choices(["shipped", "pending", "cancelled"], [85, 10, 5])[0],
                 f"2026-{random.randint(1, 9):02d}-{random.randint(1, 28):02d}") for _ in
                 range(N)))
db.execute("CREATE INDEX by_status ON orders(status)")
q = "SELECT count(*) FROM orders WHERE status = 'shipped' AND created_at >= '2026-09-01'"
print("    before ANALYZE:", [r[-1] for r in db.execute("EXPLAIN QUERY PLAN " + q)])
db.execute("ANALYZE")
print("    after ANALYZE: ", [r[-1] for r in db.execute("EXPLAIN QUERY PLAN " + q)])
print("    what it learned:", db.execute("SELECT stat FROM sqlite_stat1 WHERE idx =
    'by_status').fetchone()[0],
      "(rows, rows per distinct value)")

```

OUTPUT

```

1. skew: WHERE country = 'US'
   uniform guess N/distinct = 3,225    with most-common-values list = 70,117    actual = 70,117
2. correlation: WHERE country = 'C07' AND city = 'Paris'
   independence guess = 3    actual = 308    (every Paris row is in C07)
3. the plan follows the estimate: join those rows to a 600-page customers table
   estimated    3 outer rows: nested loop    12 hash join    600 -> nested loop
   actual    308 outer rows: nested loop    1,232 hash join    631 -> hash join
4. a real planner, before and after it gathers statistics
   before ANALYZE: ['SEARCH orders USING INDEX by_status (status=?)']
   after ANALYZE:  ['SCAN orders']
   what it learned: 100000 33334 (rows, rows per distinct value)

```

The first case is **skew**. Seventy percent of customers are in one country, but a planner that only knows the number of distinct values assumes they're spread evenly and guesses 3,225 rows where there are 70,117, an error of more than twenty times. A most-common-values list fixes it exactly, which is why databases keep one.

The second case is **correlation**. Every Paris row is in country C07, so the second condition adds no filtering at all. The planner, multiplying the two selectivities as if they were independent, estimates 3 rows; there are 308. The third block shows why that matters. At 3 rows, an index nested loop costs 12 page reads and looks far cheaper than a hash join's 600, so the planner picks it. At the real 308 rows, the nested loop costs 1,232 and the hash join would have cost 631. The plan was the right choice for the wrong number, and errors like this compound: in a query with five joins, a tenfold error at the bottom can become a millionfold error at the top.

The last block is a real planner changing its mind. Without statistics, SQLite assumes an equality condition is selective and uses the index on `status`. After `ANALYZE`, it knows the index has about 33,000 rows per distinct value, so following it would mean tens of thousands of scattered row lookups, and it switches to a plain scan. Chapter 11 measures both plans and confirms the planner is right to switch.

Why estimates go wrong

Beyond skew and correlation, the usual causes are stale statistics after a bulk load or a big delete, conditions the statistics can't see into (functions, expressions, `LIKE` patterns, values from a subquery), and **parameter sensitivity**. A prepared statement may be planned once for a generic parameter and reused for a value that matches a million rows instead of ten. Joins make all of these worse, because the error in each input's estimate multiplies through every join above it.

Fixing a bad plan

Start with `EXPLAIN ANALYZE` and compare each step's estimated rows with its actual rows. The step where they first diverge badly is where the plan went wrong, and everything above it inherited the error. Then work through the fixes in order of how well they age. Run `ANALYZE`, and make it the last step of every bulk load. Raise the statistics detail for skewed columns (PostgreSQL's per-column `STATISTICS` target). Tell the planner about correlated columns: PostgreSQL's `CREATE STATISTICS ... (dependencies)` exists for exactly the Paris case. If none of that works, rewrite the query: split it, or materialise an intermediate result whose size is then known. Hints and forced indexes come last. They fix today's plan by freezing it, and they keep forcing it after the data has changed so much that it's wrong.

COMMON TRAPS

Stale statistics after a bulk load, so the planner plans for a table a fraction of its real size.

Reading `EXPLAIN` without `ANALYZE` which shows the estimates but not how wrong they are

Reading `EXPLAIN` without `ANALYZE`, which shows the estimates but not how wrong they are.

Correlated columns treated as independent, making combined conditions look far more selective than they are.

Generic plans for skewed parameters, fast for most values and disastrous for the big one.

Hints that outlive the data they were written for.

EXERCISES

EXERCISE 10.1 Read the divergence

`EXPLAIN ANALYZE` shows a nested loop whose outer input is estimated at `rows=12` but produces `actual rows=48,213`, and the query takes 40 seconds. Diagnose it and list your next steps.

SOLUTION

The planner chose a nested loop because it expected 12 outer rows, meaning 12 inner index lookups. It got 48,213, so it did 48,213 lookups, each a random read: exactly the trap in the example. Find out why the estimate is low. Check when the table was last analysed and run `ANALYZE`; check whether the outer input's filter combines correlated columns, uses an expression, or depends on a parameter. After fixing the statistics, the planner should switch to a hash join or a different join order on its own. If the cause is correlation, add extended statistics on those columns; if it's an expression, an expression index also gives the planner statistics on that expression. Only if the estimate can't be fixed, consider restructuring the query.

EXERCISE 10.2 Slow until nine

A nightly job loads 10 million rows into a 2-million-row table. Every morning, reports are slow until about 9:00, then fine for the rest of the day. Explain the pattern and the fix.

SOLUTION

After the load, the statistics still describe a 2-million-row table, so the planner picks plans that suit one, such as nested loops and index lookups sized for small inputs, against a table six times larger. Around 9:00 the automatic analyse process notices how many rows have changed and refreshes the statistics, and the plans correct themselves.

Fix it by running `ANALYZE` on the loaded tables as the final step of the job, before anyone queries them. The general rule: any operation that changes a large fraction of a table should end with fresh statistics.

EXERCISE 10.3 Telling the planner about correlation

Queries filter on both `country` and `city`, which are strongly correlated, and their estimates are consistently far too low. Give two fixes.

SOLUTION

First, give the planner statistics on the pair. In PostgreSQL, `CREATE STATISTICS geo_dep (dependencies) ON country, city FROM customers;` followed by `ANALYZE` records that city largely determines country, so the planner stops multiplying the selectivities. Other engines approach this with multi-column statistics or histograms on a combined column. Second, change the query: if city implies country, filter on city alone (or on a city id), and the redundant condition the planner was misjudging disappears. A composite index on `(country, city)` also helps some planners, because its statistics describe the combination.

CHECKLIST

- ☐ Do you compare estimated and actual rows when a plan looks wrong?
- ☐ Does every bulk load end with `ANALYZE`?
- ☐ Do skewed columns have enough statistics detail?
- ☐ Have you told the planner about correlated columns?
- ☐ Do you reach for hints only after statistics and rewrites have failed?

GO DEEPER

REFERENCE Statistics used by the planner PostgreSQL documentation

PAPER How good are query optimizers, really? Leis et al., VLDB 2015

REFERENCE `ANALYZE` SQLite documentation

A million rows under fifty milliseconds

3 exercises

Tuning a query is an experiment, not an act of inspiration: measure, read the plan, change one thing, measure again, and write down what happened. This chapter does the stage's deliverable on a real million-row table, including the change that made things slower, because that's the one that teaches the most.

The method

Start with data that resembles production: the same row count, or close to it, and the same distributions. A query tuned on ten thousand evenly spread rows tells you nothing about ten million skewed ones. Measure each version several times and take the median, so one slow run doesn't mislead you, and note whether the cache was warm. Read the plan before changing anything, form a hypothesis about what it's doing badly, and change one thing. Then measure again, check the result is identical, and record the plan and the time. The record matters as much as the speed-up: it's how you explain the change in review, and how the next person learns why the index exists.

The query

The table holds a million orders spread over 21 months, 85% of them shipped. The report wants daily shipped revenue for September 2026, about 4.7% of the table. Here is the whole session, run against a real SQLite database. The output is one recorded run on a laptop; your timings will differ, but the ratios and the plans won't.

EXAMPLE million.py · the tuning session, plan and timing at every step

```
import random, sqlite3, statistics, time
from datetime import date, timedelta

random.seed(11)
db = sqlite3.connect(":memory:")
db.execute("""CREATE TABLE orders(id INTEGER PRIMARY KEY, customer_id INTEGER, status TEXT,
                                   created_at TEXT, total_cents INTEGER)""")

start = date(2025, 1, 1)
days = [(start + timedelta(days=d)).isoformat() for d in range(638)]
db.executemany("INSERT INTO orders(customer_id, status, created_at, total_cents) VALUES (?, ?,
?, ?)",
               ((random.randint(1, 50_000), random.choices(["shipped", "pending", "cancelled"], [85, 10,
5])[0],
random.choice(days), random.randint(500, 50_000)) for _ in range(1_000_000)))

QUERY = """SELECT created_at AS day, COUNT(*) AS orders, SUM(total_cents) AS revenue
FROM orders
WHERE status = 'shipped' AND created_at >= '2026-09-01' AND created_at < '2026-10-
01'
GROUP BY created_at"""
```

```

def measure(label):
    plan = " / ".join(r[-1] for r in db.execute("EXPLAIN QUERY PLAN " + QUERY))
    times = []
    for _ in range(5):
        t = time.perf_counter()
        result = db.execute(QUERY).fetchall()
        times.append((time.perf_counter() - t) * 1000)
    ms = statistics.median(times)
    print(f"{label}\n    plan: {plan}\n    median of 5 runs: {ms:7.1f} ms    ({len(result)} days)")
    return ms, result

print(f"{db.execute('SELECT COUNT(*) FROM orders').fetchone()[0]:,} orders; report: daily shipped revenue for September 2026\n")
base_ms, expected = measure("step 0: no index")

db.execute("CREATE INDEX by_status ON orders(status)")
ms1, r1 = measure("step 1: index on status (85% of rows are 'shipped')")

db.execute("ANALYZE")
ms2, r2 = measure("step 2: ANALYZE, so the planner knows status barely filters")

db.execute("DROP INDEX by_status")
db.execute("CREATE INDEX by_created ON orders(created_at)")
db.execute("ANALYZE")
ms3, r3 = measure("step 3: index on created_at, the selective column")

db.execute("DROP INDEX by_created")
db.execute("CREATE INDEX by_status_created_total ON orders(status, created_at, total_cents)")
db.execute("ANALYZE")
ms4, r4 = measure("step 4: covering index: equality column, then range column, then the summed column")

assert r1 == r2 == r3 == r4 == expected # same answer at every step
print(f"\nsame result at every step: True | {base_ms:.0f} ms -> {ms4:.1f} ms, "
      f"{base_ms / ms4:.0f}x faster | under 50 ms: {ms4 < 50}")

```

OUTPUT

1,000,000 orders; report: daily shipped revenue for September 2026

step 0: no index

plan: SCAN orders / USE TEMP B-TREE FOR GROUP BY
median of 5 runs: 68.2 ms (30 days)

step 1: index on status (85% of rows are 'shipped')

plan: SEARCH orders USING INDEX by_status (status=?) / USE TEMP B-TREE FOR GROUP BY
median of 5 runs: 82.4 ms (30 days)

step 2: ANALYZE, so the planner knows status barely filters

plan: SCAN orders / USE TEMP B-TREE FOR GROUP BY
median of 5 runs: 68.3 ms (30 days)

step 3: index on created_at, the selective column

plan: SEARCH orders USING INDEX by_created (created_at>? AND created_at<?)
median of 5 runs: 31.3 ms (30 days)

step 4: covering index: equality column, then range column, then the summed column

plan: SEARCH orders USING COVERING INDEX by_status_created_total (status=? AND created_at>? AND created_at<?)
median of 5 runs: 3.1 ms (30 days)

same result at every step: True | 68 ms -> 3.1 ms, 22x faster | under 50 ms: True

Reading the session

Step 0, 68.2 ms. With no index, the plan is a scan of all million rows followed by a temporary B-tree to group the results by day. That's not terrible for a scan, but it reads every page to use one row in twenty.

Step 1, 82.4 ms: slower. An index on `status` looks like the obvious move, since the query filters on it, and with no statistics the planner assumes an equality condition is selective and uses it. But 85% of rows match. Following the index means about 850,000 row lookups in index order, each landing on a scattered page, instead of one sequential pass through the table. A low-selectivity index can be worse than none, and this is what it looks like.

Step 2, 68.3 ms. `ANALYZE` tells the planner what step 1 found out the hard way: `status` barely filters. With the statistics, it abandons the index and goes back to the scan, and the time returns to the baseline. The useless index is still costing every insert and update, so the next step drops it.

Step 3, 31.3 ms. An index on `created_at`, the selective column, narrows the read to September's 47,000 or so rows. Two things changed in the plan. The `SEARCH ... (created_at>? AND created_at<?)` line shows the index is used for the range, and the temporary B-tree for grouping has disappeared: the index delivers rows already in date order, so each day's group arrives together and no sort is needed. What's left is a table lookup per row, to check `status` and read `total_cents`.

Step 4, 3.1 ms. A covering index on `(status, created_at, total_cents)` follows all of chapter 8's rules at once. The equality column comes first, so the shipped orders are contiguous; the range column comes second, so September's shipped orders are one slice; and the summed column is included, so the plan says `COVERING INDEX` and never touches the table. The answer is identical at every step, the final time is 22 times faster than the scan, and it is comfortably under the fifty-millisecond target.

Writing it down

The deliverable asks for each plan change to be documented, and a table like this is enough. It's what a reviewer needs and what the next engineer will thank you for.

STEP	CHANGE	PLAN	MEDIAN	LESSON
0	none	scan + temp B-tree for grouping	68.2 ms	baseline reads every page
1	index on <code>status</code>	index search + grouping	82.4 ms	85% selectivity: worse than a scan
2	<code>ANALYZE</code>	scan + grouping	68.3 ms	statistics fix the plan, not the index
3	index on <code>created_at</code>	index range search, no sort	31.3 ms	selective column; order removes the sort

4	covering (status, created_at, total_cents)	covering index slice	3.1 ms	equality, range, included: no table reads
---	---	----------------------	-----------	--

What it costs, and what else would work

The covering index isn't free. It is a second copy of three columns for every row, and every insert, and every update to those columns, now maintains another B-tree. For a report on a dashboard that refreshes every minute, that's a good trade. For a report run once a month, a slower query is fine and the index is waste. Indexes also aren't the only tool. A **summary table** of revenue per day, updated as orders ship, answers the report by reading thirty rows. A **materialised view** does the same with the refresh managed by the database. **Partitioning** the orders table by month lets the planner skip every partition but September's. And a cache in front of the database (chapter 19) can skip the query altogether. Pick by how fresh the answer must be and how often it's asked.

COMMON TRAPS

Tuning on a toy dataset, a thousand times smaller and evenly distributed.

Measuring once, so a cold cache or a background job decides the result.

Changing several things at once, so you can't tell which change helped and which hurt.

Adding an index on the column in the WHERE clause without checking how selective it is.

Never checking the results are identical after a rewrite.

EXERCISES

EXERCISE 11.1 Is the index worth it?

The step 4 index makes order inserts about 15% slower. The report behind it runs on an internal dashboard that ten people open a few times a day. Keep the index or not? What would change your answer?

SOLUTION

Probably not. A few dozen runs a day at 68 ms cost a couple of seconds in total, while a 15% insert slowdown is paid on every order, all day, and gets worse as more indexes accumulate. Drop the covering index; if the dashboard feels slow, the step 3 index alone gets it to about 31 ms with less maintenance, or a daily summary table makes it instant at almost no write cost. The answer flips if the report becomes customer-

facing or runs constantly (an API endpoint, a page every merchant opens), or if inserts aren't the bottleneck. Decide from measured frequencies on both sides, not from the speed-up alone.

EXERCISE 11.2 Where did the sort go?

Steps 0 to 2 show `USE TEMP B-TREE FOR GROUP BY`; steps 3 and 4 don't. Explain why, and name another query shape that benefits the same way.

SOLUTION

To group rows by day, the engine needs each day's rows together. A scan returns rows in storage order, so the engine builds a temporary B-tree keyed by day and groups from that. An index on `created_at` returns rows already sorted by date, so each day's rows arrive consecutively and can be aggregated on the fly, with no sort. In step 4 the index is sorted by `status` first, but within the single value `'shipped'` it's sorted by date, so the same holds. Any `ORDER BY` or `GROUP BY` that matches an index's order benefits, most visibly `ORDER BY created_at DESC LIMIT 20`, which becomes "read twenty index entries and stop".

EXERCISE 11.3 Plan a second tuning session

Another endpoint runs `SELECT id, created_at, total_cents FROM orders WHERE customer_id = ? ORDER BY created_at DESC LIMIT 20`. Predict the baseline plan, choose an index, and predict the plan after it.

SOLUTION

Baseline: `SCAN orders` plus `USE TEMP B-TREE FOR ORDER BY`, reading a million rows to sort one customer's twenty or so. Choose `(customer_id, created_at, total_cents)`: equality column first, then the sort column, then the remaining selected column; `id` is the rowid, which every SQLite index already carries. The expected plan is `SEARCH orders USING COVERING INDEX ... (customer_id=?)` with no temporary B-tree, because the engine reads that customer's slice backwards from the newest entry and stops after twenty. Expect well under a millisecond. Then do what this chapter did: measure, record the plan, and check the result matches.

- ☐ Do you tune against production-like volume and distributions?
- ☐ Do you take the median of several runs and note cache state?
- ☐ Do you change one thing at a time and confirm identical results?
- ☐ Is every change recorded with its plan and timing?
- ☐ Have you weighed the index's write cost against how often the query runs?

GO DEEPER

GUIDE [Testing and scalability](#) Markus Winand, [Use The Index](#), Luke

TOOL explain.depesz.com a PostgreSQL plan visualiser

REFERENCE [EXPLAIN QUERY PLAN](#) SQLite documentation

PART IV

Transactions

Four chapters on many people at once and power that fails. What each letter of ACID really promises, the anomalies each isolation level lets through, the locks and versions that implement isolation, and the write-ahead log that makes a commit survive a crash. Chapter 15 builds the first half of the stage's deliverable.

What ACID really promises

3 exercises

ACID names four promises a transaction makes, and they're less uniform than the acronym suggests. Atomicity and durability are the database's job, and it does them well. Isolation comes in weaker forms than its name implies (chapter 13). And consistency is mostly your job, done with constraints and with transactions drawn around the right operations.

A transaction is the unit of every promise

A transaction is a group of statements between `BEGIN` and `COMMIT` that the database treats as one indivisible unit. Every ACID promise is made per transaction, which makes it important to know where your transactions actually begin and end. Many drivers default to `autocommit`, where every statement is its own transaction, so a two-statement operation is two separate promises, not one. Others open transactions implicitly and leave them open until you commit, as chapter 2's foreign-key trap showed. The example runs the classic test, a transfer that dies halfway, both ways.

EXAMPLE `acid.py` · a transfer that crashes halfway, with and without a transaction

```
import os, sqlite3, tempfile

path = os.path.join(tempfile.mkdtemp(), "bank.db")
db = sqlite3.connect(path, isolation_level=None)          # autocommit unless we say BEGIN
db.executescript("""
CREATE TABLE accounts(id TEXT PRIMARY KEY, balance INTEGER NOT NULL CHECK (balance >= 0));
INSERT INTO accounts VALUES ('alice', 100), ('bob', 50);
""")

def balances():
    rows = dict(db.execute("SELECT id, balance FROM accounts ORDER BY id"))
    return f"{rows} total {sum(rows.values())}"

def reset():
    db.execute("UPDATE accounts SET balance = CASE id WHEN 'alice' THEN 100 ELSE 50 END")

class Crash(Exception):
    pass

def transfer(src, dst, amount, crash=False, atomic=True):
    if atomic:
        db.execute("BEGIN")
    try:
        db.execute("UPDATE accounts SET balance = balance - ? WHERE id = ?", (amount, src))
        if crash:
            raise Crash("the process died between the two updates")
        db.execute("UPDATE accounts SET balance = balance + ? WHERE id = ?", (amount, dst))
        if atomic:
            db.execute("COMMIT")
    return "ok"
```

```

except (Crash, sqlite3.IntegrityError) as e:
    if atomic:
        db.execute("ROLLBACK")
    return f"failed ({e})"

print("start:                                ", balances())
print("A. no transaction, crash:             ", transfer("alice", "bob", 30, crash=True, atomic=False))
print("                                       ", balances(), "<- 30 vanished")
reset()
print("B. transaction, crash:                 ", transfer("alice", "bob", 30, crash=True))
print("                                       ", balances())
print("C. transaction, overdraft:              ", transfer("bob", "alice", 500))
print("                                       ", balances())

db.execute("BEGIN")
db.execute("UPDATE accounts SET balance = balance - 10 WHERE id = 'alice'")
db.execute("SAVEPOINT bonus")
db.execute("UPDATE accounts SET balance = balance + 999 WHERE id = 'bob'")
db.execute("ROLLBACK TO bonus")                # undo only the work since the
                                              savepoint
db.execute("UPDATE accounts SET balance = balance + 10 WHERE id = 'bob'")
db.execute("COMMIT")
print("D. savepoint undid the bonus:", balances())

db.close()
reopened = sqlite3.connect(path)
print("E. after reopening the file: ", dict(reopened.execute("SELECT id, balance FROM accounts
ORDER BY id")))

```

OUTPUT

```

start:                                {'alice': 100, 'bob': 50} total 150
A. no transaction, crash:             failed (the process died between the two updates)
                                       {'alice': 70, 'bob': 50} total 120 <- 30 vanished
B. transaction, crash:                 failed (the process died between the two updates)
                                       {'alice': 100, 'bob': 50} total 150
C. transaction, overdraft:             failed (CHECK constraint failed: balance >= 0)
                                       {'alice': 100, 'bob': 50} total 150
D. savepoint undid the bonus:          {'alice': 90, 'bob': 60} total 150
E. after reopening the file:          {'alice': 90, 'bob': 60}

```

Atomicity: all or nothing

Case A is the bug every financial system fears. Without a transaction, the debit committed on its own and the credit never happened, so 30 left the system. Case B runs the same crash inside a transaction: the rollback undoes the debit and the books still balance. The database delivers this by recording how to undo each change until the commit. SQLite keeps a rollback journal or, in WAL mode, doesn't touch the main file until commit; PostgreSQL keeps old row versions; InnoDB writes undo records. However it's done, the promise is the same: a failed transaction leaves no trace.

Consistency: the rules the database knows

Case C is consistency as the database understands it: the transfer would have made Bob's balance negative, the `CHECK` constraint refused it, and the whole transaction rolled back. That's the limit of the database's version of consistency: it enforces the constraints you declared and nothing else. The rule that the total of all balances never changes in a transfer

is consistency too, but no constraint expresses it. It holds only because the debit and credit are in one transaction. That's why the C in ACID is the application's responsibility, and why chapter 2 pushed so hard on declaring constraints.

Isolation, briefly

Isolation promises that concurrent transactions don't see each other's half-finished work. Fully honoured, it means they behave as if they ran one at a time, a level called **serializable**. Most databases default to weaker levels that are faster and allow specific anomalies, and chapter 13 shows exactly which. The savepoint in case D is a smaller tool: a named point inside a transaction you can roll back to without abandoning the whole thing, which is useful when one step of a larger operation may fail and be skipped.

Durability: committed means committed

Case E reopens the database file: the committed transfer and the savepoint result are there, because a commit isn't acknowledged until its log record is flushed to disk (chapter 15). Durability can be weakened deliberately for speed. PostgreSQL's `synchronous_commit = off`, MySQL's `innodb_flush_log_at_trx_commit = 2` and SQLite's `synchronous = NORMAL` each acknowledge commits before the flush, risking the last fraction of a second of commits in a crash, without corrupting anything. Hardware can weaken it accidentally: a disk with a volatile write cache that reports a flush as done before it is. Servers use drives with power-loss protection for exactly this reason.

Drawing the boundaries

Draw a transaction around one business operation: everything that must happen together, and nothing else. Keep it short. A transaction holds locks and an old snapshot for its whole life, and a long one blocks other writers and holds back the cleanup chapter 14 describes. Above all, never make a network call to another service inside a transaction. The call may take seconds while every lock waits, and if the transaction rolls back afterwards, the other service's action can't be undone with it. Make the external call outside, with an idempotency key so a retry can't do it twice. Across services there is no ACID at all; Fleet's sagas and outbox pattern take over there.

COMMON TRAPS

Autocommit for multi-step operations, turning one promise into several that can fail independently.

Calling external APIs inside a transaction, holding locks during the call and leaving actions that can't be rolled back.

Catching an exception and committing anyway, keeping half of the work.

Long or idle transactions that hold locks and old snapshots for minutes.

Retrying without idempotency, so a transaction that actually committed runs twice.

EXERCISES

EXERCISE 12.1 Draw the boundaries

A checkout inserts an order, decrements stock, charges the card through a payment API, and marks the order paid. Where should the transaction boundaries go?

SOLUTION

Not around all four steps, because the payment call would sit inside the transaction. Use two short transactions with the call between them. Transaction 1 inserts the order as `pending` and decrements stock, which reserves the goods atomically. Then call the payment API outside any transaction, with an idempotency key such as the order id, so a retry after a timeout can't charge twice. Transaction 2 marks the order `paid`. If the payment fails, a compensating transaction cancels the order and restores the stock, and a periodic job expires orders left pending, which covers a crash between the call and transaction 2.

EXERCISE 12.2 Name the letter

Which ACID property failed? (a) After a crash between two statements, the sum of all balances changed. (b) An order confirmed to the customer is missing after a power cut. (c) A report shows money that has left one account but not yet arrived in the other. (d) Stock went negative because of an application bug.

SOLUTION

- a. **Atomicity**, as in case A. The statements weren't in one transaction, or the transaction wasn't rolled back.
- b. **Durability**. The commit was acknowledged before the log reached stable storage, through a relaxed setting or a disk cache that lied about flushing.
- c. **Isolation**. The report saw an intermediate state. At any standard isolation level above read uncommitted, it shouldn't see uncommitted changes, so suspect a report that combines the results of separate transactions.

- d. **Consistency**, the application's share of it. Declare `CHECK (stock >= 0)` and the database turns the bug into a refused transaction instead of bad data.

EXERCISE 12.3 Trading durability for speed

Turning on PostgreSQL's `synchronous_commit = off` triples write throughput in your benchmark. What exactly can you lose, and for which data is it acceptable?

SOLUTION

Commits are acknowledged before their log records are flushed; the flush happens shortly afterwards in the background (within about three times `wal_writer_delay`, well under a second by default). If the server crashes in that window, the most recent acknowledged transactions are lost, but the database stays consistent, because no transaction is ever half-applied. That's acceptable for data you can afford to lose a second of: analytics events, view counters, logs, caches. It isn't acceptable for money, orders or anything a user was told was saved. PostgreSQL lets you set it per transaction, so you can relax it for the event stream and keep it on for payments in the same database.

CHECKLIST

- ☐ Do you know whether your driver runs in autocommit mode?
- ☐ Is every multi-step business operation inside one transaction?
- ☐ Are transactions short, with no external calls inside them?
- ☐ Are invariants beyond constraints protected by transaction boundaries?
- ☐ Do you know your durability settings and what they risk?

GO DEEPER

PAPER The Transaction Concept: Virtues and Limitations Jim Gray, 1981

REFERENCE Transactions SQLite documentation

REFERENCE Asynchronous commit PostgreSQL documentation

Isolation levels and their anomalies

3 exercises

Almost no database gives you full isolation by default. Each weaker level lets through specific anomalies, and the bugs they cause look like ordinary logic errors that only appear under load. Know which level you're running, which anomalies it allows, and which of your operations can't tolerate them.

Levels defined by what goes wrong

Serializable isolation means concurrent transactions produce the same result as running them one at a time in some order. It's the only level at which you can reason about each transaction alone. The weaker levels exist because serializability costs throughput, and the SQL standard defines them by the anomalies they forbid. **Read uncommitted** allows dirty reads of another transaction's uncommitted changes. **Read committed** forbids those, but allows a value to change between two reads in the same transaction. **Repeatable read** keeps such values stable, but in the standard's definition allows phantom rows to appear. A famous 1995 paper by Berenson and colleagues showed these definitions miss two anomalies that matter in practice: the **lost update** and **write skew**.

Defaults vary, and the names don't mean the same thing everywhere. PostgreSQL, Oracle and SQL Server default to read committed; MySQL's InnoDB defaults to repeatable read. PostgreSQL's repeatable read is really **snapshot isolation**: each transaction reads from a snapshot taken when it starts. Its serializable level adds conflict detection on top of snapshots (serializable snapshot isolation, since version 9.1). SQLite is serializable because it allows only one writer at a time. The example implements three levels in a small multi-version engine and runs the three classic scenarios against each.

EXAMPLE isolation.py · three scenarios at three isolation levels

```
class Database:
    """A tiny multi-version store: every committed write adds a version stamped with a commit
    time."""
    def __init__(self, data):
        self.versions = {k: [(0, v)] for k, v in data.items()}
        self.clock = 0
    def value(self, key, as_of):
        return [v for ts, v in self.versions[key] if ts <= as_of][-1]
    def begin(self, level):
        return Transaction(self, level)

class Transaction:
    def __init__(self, db, level):
        self.db, self.level, self.start = db, level, db.clock
        self.reads, self.writes, self.status = set(), {}, "active"
    def read(self, key):
        self.reads.add(key)
        if key in self.writes:
```

```

        return self.writes[key]
        # read committed sees the latest commit; the other two levels read from their start
        snapshot
        return self.db.value(key, self.db.clock if self.level == "read committed" else
                               self.start)
    def write(self, key, value):
        self.writes[key] = value
    def commit(self):
        changed = {k for k, vs in self.db.versions.items() if vs[-1][0] > self.start}
        if self.level != "read committed" and changed & set(self.writes):
            self.status = "aborted: someone else wrote that row first"
            return
        if self.level == "serializable" and self.writes and changed & self.reads:
            self.status = "aborted: data it read has changed"
            return
        self.db.clock += 1
        for k, v in self.writes.items():
            self.db.versions[k].append((self.db.clock, v))
        self.status = "committed"

def lost_update(level):
    db = Database({"counter": 0})
    a, b = db.begin(level), db.begin(level)
    x, y = a.read("counter"), b.read("counter")           # both read 0
    a.write("counter", x + 1); b.write("counter", y + 1)
    a.commit(); b.commit()
    if b.status == "committed":
        note = " (both committed: one increment lost)"
    else:
        note = f" (B {b.status}; it can retry)"
    return f"counter = {db.value('counter', db.clock)}{note}"

def write_skew(level):
    db = Database({"alice": 1, "bob": 1})                 # rule: at least one doctor on call
    a, b = db.begin(level), db.begin(level)
    if a.read("alice") + a.read("bob") >= 2: a.write("alice", 0)
    if b.read("alice") + b.read("bob") >= 2: b.write("bob", 0)
    a.commit(); b.commit()
    on_call = db.value("alice", db.clock) + db.value("bob", db.clock)
    note = " (both committed: the rule is broken)" if b.status == "committed" else f" (B
        {b.status})"
    return f"{on_call} doctor(s) on call{note}"

def non_repeatable(level):
    db = Database({"price": 100})
    t1 = db.begin(level)
    first = t1.read("price")
    t2 = db.begin(level); t2.write("price", 70); t2.commit()
    second = t1.read("price"); t1.commit()
    return f"T1 read {first}, then {second}"

for name, scenario in (("lost update: two increments", lost_update),
                       ("write skew: both go off call", write_skew),
                       ("non-repeatable read", non_repeatable)):
    print(name)
    for level in ("read committed", "snapshot", "serializable"):
        print(f"    {level:15} {scenario(level)}")

```

OUTPUT

```

lost update: two increments
  read committed  counter = 1 (both committed: one increment lost)
  snapshot        counter = 1 (B aborted: someone else wrote that row first; it can retry)
  serializable    counter = 1 (B aborted: someone else wrote that row first; it can retry)
write skew: both go off call
  read committed  0 doctor(s) on call (both committed: the rule is broken)

```

snapshot	0 doctor(s) on call (both committed: the rule is broken)
serializable	1 doctor(s) on call (B aborted: data it read has changed)
non-repeatable read	
read committed	T1 read 100, then 70
snapshot	T1 read 100, then 100
serializable	T1 read 100, then 100

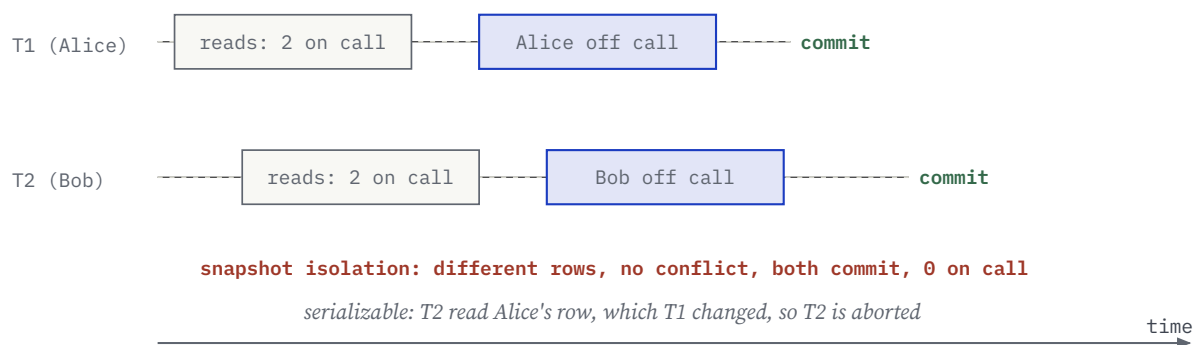
The lost update

Two transactions each read a counter, add one in application code and write it back. At read committed, both read 0, both write 1, and both commit: one increment is gone and nothing reported an error. Snapshot isolation catches it, because the second transaction tries to write a row that someone else changed since its snapshot, and it aborts with an error the application can retry. Note the shape of the bug: a read, then a write computed from it.

`UPDATE counters SET n = n + 1` is a single atomic statement, and every common database executes it safely at read committed. The lost update lives in read-modify-write code, which is exactly what ORMs generate when you load an object, change a field and save it.

Write skew

The on-call rule says at least one doctor must stay on call. Alice and Bob both feel unwell at the same moment, and each transaction checks the rule: two doctors on call, so it's safe for me to go. Alice's transaction updates Alice's row; Bob's updates Bob's. Neither writes a row the other wrote, so snapshot isolation sees no conflict, both commit, and nobody is on call. Each transaction was correct on its own; together they broke the rule. Only serializable catches it here, by noticing that Bob's transaction read a row (Alice's) that changed before it committed.



Write skew. Each transaction checks the rule against its own snapshot and writes a different row, so neither sees the other's change. The rule is about the combination of rows, which no single write conflicts on.

The non-repeatable read is the mildest of the three: at read committed, a transaction that reads the price twice sees 100 and then 70, because another transaction committed in between. A report that sums one table and then another can be inconsistent this way.

Snapshot isolation and serializable both read from the transaction's starting snapshot, so the price stays 100 until the transaction ends.

Defences short of serializable

You don't have to switch the whole database to serializable to be safe; you have to protect the operations that can't tolerate an anomaly. Replace read-modify-write with a single atomic statement where you can. Lock what you read with `SELECT ... FOR UPDATE`, so concurrent transactions queue: in the on-call case, locking both doctors' rows before checking makes the second transaction wait and then see the first one's change. Use optimistic version checks (chapter 14). Turn the rule into a constraint where it fits one, or **materialise the conflict**: give the rule a row of its own, such as one row per shift, that every transaction touching the rule must update, so they conflict on it. Or run serializable for those transactions and retry when they abort, as exercise 13.3 describes.

Isolation is subtle enough that database implementations get it wrong too. Kyle Kingsbury's Jepsen analyses test databases against their documented guarantees and have found real violations in many of them, including a serializable-isolation bug in PostgreSQL in 2020, fixed promptly after the report.

COMMON TRAPS

Assuming you're running serializable when the default is read committed or snapshot.

Read-modify-write in application code, the source of lost updates at read committed.

Check-then-act on a rule spanning several rows, which snapshot isolation lets break through write skew.

Raising the isolation level without retry logic, so serialization failures become user-facing errors.

Trusting a level's name without reading what your engine means by it.

EXERCISES

EXERCISE 13.1 Name the anomaly

(a) Two customers both book the last seat on a flight; the code counts bookings, then inserts one if there's room. (b) A report's total changes between two queries in the same transaction. (c) Two requests decrement stock from 5, and it ends at 4.

SOLUTION

- a. **Write skew**, in its phantom form: each transaction counted the bookings, saw room, and inserted a new row that the other's count didn't include. The writes don't conflict, because they're different new rows.
- b. **Non-repeatable read**, at read committed. Run the report at repeatable read or snapshot isolation so it reads one snapshot throughout.
- c. **Lost update**. Both read 5, both wrote 4. Use `UPDATE ... SET stock = stock - 1 WHERE id = ? AND stock > 0` and check the affected row count.

EXERCISE 13.2 Fix the on-call bug at read committed

Without changing the isolation level, rewrite the "go off call" transaction so the rule can't be broken.

SOLUTION

Lock the rows the rule depends on before checking it: `SELECT id FROM doctors WHERE shift_id = ? AND on_call FOR UPDATE`, then count them, then update your own row if the count is at least two. The second transaction blocks on the lock until the first commits, then re-reads and sees one doctor on call, so it refuses. A single conditional statement also works: `UPDATE doctors SET on_call = false WHERE id = :me AND (SELECT count(*) FROM doctors WHERE shift_id = :s AND on_call) >= 2`. But that relies on the engine's exact read-committed behaviour for subqueries, so the explicit lock is clearer. Materialising the conflict, a row per shift that every change updates, makes the transactions conflict even at snapshot isolation.

EXERCISE 13.3 Living with serializable

You switch a set of transactions to `SERIALIZABLE`. What must the application now do that it didn't before?

SOLUTION

Retry. A serializable database resolves conflicts by aborting one of the transactions with a serialization failure (SQLSTATE `40001` in PostgreSQL), and that's normal operation, not an error to show the user. Wrap each transaction in a loop that retries the whole transaction, not just the failed statement, a few times with randomised backoff. Make the transaction safe to retry: no external side effects inside it, and

idempotency for anything that leaves the database. Keep it short, because the longer it runs, the more likely it is to conflict. And measure the abort rate, since high contention on the same rows will show up as retries.

CHECKLIST

- ☐ Do you know your database's default isolation level and what it really means?
- ☐ Have you found the read-modify-write code paths and made them atomic?
- ☐ Are rules that span several rows protected by locks, constraints or serializable?
- ☐ Do serializable transactions retry on serialization failure?
- ☐ Could you explain write skew to a colleague with an example from your own system?

GO DEEPER

PAPER [A Critique of ANSI SQL Isolation Levels Berenson et al., 1995](#)

REFERENCE [Consistency models Jepsen](#)

REFERENCE [Transaction isolation PostgreSQL documentation](#)

Locks, MVCC and deadlocks

3 exercises

There are two ways to stop concurrent transactions trampling each other: make them wait, with locks, or let them work on different versions of the data, with multi-version concurrency control (MVCC). Every major database uses both: versions so that readers and writers never block each other, locks so that two writers can't overwrite each other. Each has a failure mode worth knowing in advance. Versions pile up, and locks deadlock.

Locks and two-phase locking

A lock is permission to use a piece of data. A **shared** lock lets you read and lets others read too; an **exclusive** lock lets you write and keeps everyone else out. The classic protocol for serializability is **two-phase locking**: a transaction acquires locks as it goes and releases none until it is done with all of them. The strict form, holding every lock until commit, is the one databases use, because it also prevents others from reading uncommitted data. It works, and it's expensive: a long report holding shared locks blocks every writer, and every writer blocks the report. Locks can cover a row, a page or a whole table. Some engines escalate many row locks into one table lock to save memory, which can suddenly block unrelated work.

MVCC: readers don't wait

MVCC removes the reader-writer conflict by never overwriting a row in place. An update creates a new version and marks the old one as ended, and each transaction reads whichever version was committed as of its snapshot. PostgreSQL stores every version in the table, stamped with the id of the transaction that created it (`xmin`) and the one that ended it (`xmax`). InnoDB and Oracle keep only the newest version in the table and reconstruct older ones from their undo logs on demand. Either way, the result is the snapshot isolation of chapter 13: a reader sees a consistent picture and never waits for a writer.

EXAMPLE `mvcc.py` · version visibility, and a deadlock caught as a cycle

```
class Version:
    def __init__(self, value, xmin):
        self.value, self.xmin, self.xmax = value, xmin, None  # created by xmin, ended by xmax

class Store:
    """Row versions with PostgreSQL-style visibility: no reader ever waits for a writer."""
    def __init__(self):
        self.next_xid, self.active, self.committed, self.chain = 1, set(), set(), []
    def begin(self):
        xid = self.next_xid; self.next_xid += 1; self.active.add(xid)
```

```

        return xid, (xid, frozenset(self.active - {xid})) # snapshot: my id, who was in
        flight
    def commit(self, xid):
        self.active.discard(xid); self.committed.add(xid)
    def _done_before(self, xid, snap):
        me, in_flight = snap
        return xid == me or (xid in self.committed and xid < me and xid not in in_flight)
    def read(self, snap):
        for v in self.chain:
            if self._done_before(v.xmin, snap) and not (v.xmax and self._done_before(v.xmax,
                snap)):
                return v.value
    def write(self, xid, value):
        for v in self.chain:
            if v.xmax is None:
                v.xmax = xid # the old version is ended, not
                erased
        self.chain.append(Version(value, xid))
    def vacuum(self):
        oldest = min(self.active, default=self.next_xid)
        before = len(self.chain)
        self.chain = [v for v in self.chain if not (v.xmax in self.committed and v.xmax <
            oldest)]
        return before - len(self.chain)

s = Store()
t1, snap1 = s.begin(); s.write(t1, "balance 100"); s.commit(t1)
reader, rsnap = s.begin() # a long report starts
writer, wsnap = s.begin(); s.write(writer, "balance 70") # an update, not yet committed
print("writer sees its own change: ", s.read(wsnap))
print("reader sees the committed value:", s.read(rsnap), "(and did not wait)")
s.commit(writer)
print("after the writer commits:")
print("  reader, same snapshot: ", s.read(rsnap))
late, lsnap = s.begin()
print("  a transaction that starts now:", s.read(lsnap))
print("versions:", [(v.value, f"xmin={v.xmin}", f"xmax={v.xmax}") for v in s.chain])
print("vacuum while the report still runs removes", s.vacuum(), "versions")
s.commit(reader); s.commit(late)
print("vacuum after it finishes removes", s.vacuum(), "->", [v.value for v in s.chain])

# Deadlocks: who waits for whom, and a cycle means nobody can ever proceed.
class LockManager:
    def __init__(self):
        self.owner, self.waits_for = {}, {}
    def lock(self, txn, row):
        holder = self.owner.get(row)
        if holder in (None, txn):
            self.owner[row] = txn
            return f"T{txn} locks {row}"
        self.waits_for[txn] = holder
        cycle = self._cycle(txn)
        if cycle:
            victim = max(cycle) # abort the youngest transaction
            self.abort(victim)
            return f"T{txn} waits for T{holder} -> deadlock {' -> '.join(f'T{t}' for t in cycle
                + [cycle[0]])}; abort T{victim}"
            return f"T{txn} waits for T{holder} (holds {row})"
    def _cycle(self, start):
        path, t = [start], self.waits_for.get(start)
        while t is not None and t not in path:
            path.append(t); t = self.waits_for.get(t)
        return path if t == start else None
    def abort(self, txn):
        self.waits_for.pop(txn, None)
        for row, holder in list(self.owner.items()):

```

```

        if holder == txn:
            del self.owner[row]
        for t, h in list(self.waits_for.items()):
            proceed
            if h == txn:
                del self.waits_for[t]

print()
lm = LockManager()
for txn, row in ((1, "account A"), (2, "account B"), (1, "account B"), (2, "account A")):
    print(lm.lock(txn, row))
print(lm.lock(1, "account B"), "(T1 retries and proceeds)")

```

OUTPUT

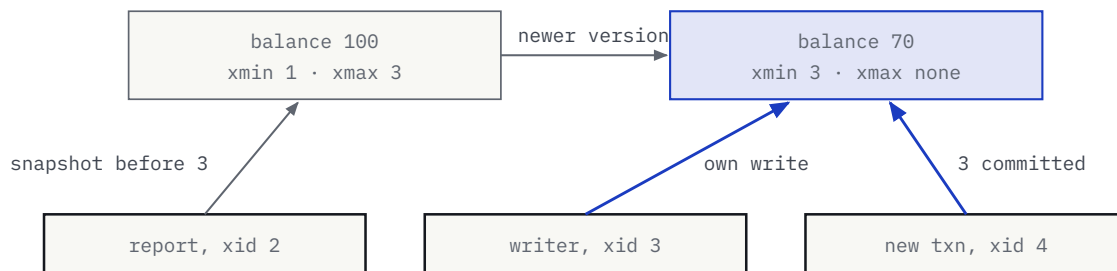
```

writer sees its own change:      balance 70
reader sees the committed value: balance 100 (and did not wait)
after the writer commits:
    reader, same snapshot:      balance 100
    a transaction that starts now: balance 70
versions: [('balance 100', 'xmin=1', 'xmax=3'), ('balance 70', 'xmin=3', 'xmax=None')]
vacuum while the report still runs removes 0 versions
vacuum after it finishes removes 1 -> ['balance 70']

T1 locks account A
T2 locks account B
T1 waits for T2 (holds account B)
T2 waits for T1 -> deadlock T2 -> T1 -> T2; abort T2
T1 locks account B (T1 retries and proceeds)

```

The first half is PostgreSQL's design in miniature. Transaction 3 updates the balance but hasn't committed: it sees its own 70, while the long report, transaction 2, sees the committed 100, without waiting. After the commit, the report still sees 100, because its snapshot predates transaction 3. A transaction that starts afterwards sees 70. The version list shows how: the old version wasn't erased, only ended (`xmax=3`), and the new one carries `xmin=3` .



the old version can be vacuumed once no snapshot from before xid 3 is still running

MVCC in PostgreSQL's style. An update ends the old version and adds a new one. Each transaction's snapshot decides which version it sees, so the report keeps reading 100 while the writer works, and nobody waits.

The cost of keeping versions

Old versions must eventually be removed, by PostgreSQL's `VACUUM` or InnoDB's purge thread, and they can only be removed once no running transaction might still need them. The example's first vacuum, run while the report is open, removes nothing; after it finishes, the dead version goes. That's why a single long transaction is dangerous on a busy database. A report running for six hours, or a session left "idle in transaction", holds back cleanup for the whole database, and tables bloat with versions nobody can see. PostgreSQL has a sharper version of the same problem. Transaction ids are 32-bit counters, and if old rows aren't vacuumed in time, the counter approaches **wraparound**, at which point the database stops accepting writes rather than risk data appearing to come from the future. Sentry in 2015 and Mailchimp's Mandrill in 2019 both had outages for exactly this reason.

Version design also shapes write costs. Because PostgreSQL writes a new version for every update, the new version needs new entries in every index on the table, unless the update qualifies as a "heap-only tuple" update that stays on the same page and touches no indexed column. Uber cited that write amplification, among other reasons, when it moved from PostgreSQL to MySQL in 2016.

Deadlocks

MVCC handles readers; writers still lock. Transaction 1 locks account A and transaction 2 locks account B. Then 1 asks for B and waits for 2, and 2 asks for A and waits for 1. Neither can ever proceed. The second half of the example catches this the way databases do: it keeps a **wait-for graph** of who is waiting for whom, and a cycle in that graph is a deadlock. The database aborts one transaction, usually the youngest or the one that has done the least work, and the other proceeds. PostgreSQL checks for cycles after a transaction has waited for `deadlock_timeout` (one second by default); InnoDB checks as each lock wait begins. The aborted transaction gets an error, and the application should retry it. Prevention is cheaper than detection: acquire locks in a consistent order, such as always locking the lower account id first, and the cycle can't form.

Optimistic concurrency

When conflicts are rare, it's cheaper to check than to lock. Give the row a `version` column, read it with the data, and write with `UPDATE ... SET ..., version = version + 1 WHERE id = ? AND version = ?`. If another writer got there first, the version won't match, the update changes zero rows, and the application reloads and retries or asks the user. This is the right tool for edits that span human time, such as a form open for ten minutes, where holding a database lock is impossible.

COMMON TRAPS

Long or idle-in-transaction sessions that hold back vacuum for the whole database.

Ignoring transaction-id wraparound warnings until the database stops accepting writes.

Locking rows in inconsistent orders across code paths, the recipe for deadlocks.

Treating a deadlock error as a bug report instead of retrying the transaction.

Holding database locks across user think time instead of using optimistic version checks.

EXERCISES

EXERCISE 14.1 The six-hour report

A monthly report runs for six hours on a PostgreSQL primary. During it, the busiest table grows by 40 GB and queries slow down. Explain the mechanism and two fixes.

SOLUTION

The report's snapshot is six hours old, and vacuum can't remove any row version that snapshot might still see, which means any version ended in the last six hours. Every update on the busy table during that time leaves a dead version that stays, so the table and its indexes bloat by 40 GB, and scans read more pages to find live rows. Fixes: run the report on a read replica, so the primary's cleanup isn't held back (configure the replica so it doesn't feed its old snapshots back to the primary, at the cost of the report occasionally being cancelled); split the report into shorter transactions over ranges of data; or export a consistent snapshot to an analytical store (chapter 20) and run it there. Watch `pg_stat_activity` for old transactions as a matter of routine.

EXERCISE 14.2 Deadlocking transfers

Transfers lock the source account, then the destination. Two transfers between the same accounts in opposite directions occasionally deadlock. Fix it without serialising all transfers.

SOLUTION

Lock in a consistent order that doesn't depend on the direction of the transfer: always lock the account with the lower id first, then the higher. A transfer from A to B and one from B to A both lock A first, so the second simply waits for the first, and no cycle can form. One statement can do it: `SELECT ... FROM accounts WHERE id IN (?, ?) ORDER BY id FOR UPDATE`. Transfers between unrelated accounts still run in parallel. Keep the deadlock retry anyway, since other code paths may lock these rows too.

EXERCISE 14.3 Two editors, one description

Two staff members open the same product description, edit it for several minutes, and save. Today the second save silently overwrites the first. Design optimistic concurrency for it.

SOLUTION

Add `version INTEGER NOT NULL DEFAULT 1` to `products`. Include the version in the edit form when it loads. On save, run `UPDATE products SET description = ?, version = version + 1 WHERE id = ? AND version = ?` with the version the form loaded. One row changed means the save succeeded. Zero rows means someone saved in between: reload the current description, show both versions (or a diff) to the user, and let them merge and save again. No lock is held while people type, and no edit is ever lost silently.

CHECKLIST

- ☐ Do you know how your engine keeps old versions and how they're cleaned up?
- ☐ Do you monitor for long-running and idle transactions?
- ☐ On PostgreSQL, do you monitor transaction-id age?
- ☐ Do all code paths lock shared rows in the same order?
- ☐ Do edits that span human time use optimistic version checks?

GO DEEPER

REFERENCE [Concurrency control PostgreSQL documentation](#)

CASE STUDY [Why Uber Engineering switched from Postgres to MySQL Uber engineering](#)

POST-MORTEM [Transaction ID wraparound in Postgres Sentry](#)

The write-ahead log and crash recovery

3 exercises

The write-ahead log is the ledger at the heart of a database: every change is described in the log, and the log is forced to disk, before the change is allowed to count. That one rule makes commits fast, makes crashes survivable, and makes replication and point-in-time recovery possible. This chapter builds a key-value store around it, the first half of the stage's deliverable.

The rule

Writing a changed page back to its place in a data file is a random write, and a transaction may change dozens of pages scattered across many files. Making all of them durable before acknowledging a commit would be slow, and a crash partway through would leave some pages old and some new. The write-ahead log solves both problems with one rule: **before any change reaches the data files, a record describing it must be on disk in the log**. A commit then needs only one sequential append and one flush. The changed pages stay in the buffer pool and are written back later, at leisure, as chapter 1's diagram showed. After a crash, recovery replays the log: committed changes whose pages never reached disk are **redone**, and changes from transactions that never committed, which may have reached disk early, are **undone**. Production databases implement this with the ARIES algorithm, published by IBM in 1992. The store below needs only the redo half, because it never writes uncommitted data anywhere but the log.

EXAMPLE wal.py · a key-value store with a write-ahead log, checkpoints and group commit

```
import json, os, struct, tempfile, zlib
from contextlib import contextmanager

class KVStore:
    """A key-value store made durable by a write-ahead log: every change is appended and
    flushed to the log before it is applied, so the log can rebuild the data after a crash."""
    HEADER = struct.Struct("<II") # payload length, CRC-32 of the
    payload

    def __init__(self, directory):
        self.log_path = os.path.join(directory, "wal.log")
        self.snap_path = os.path.join(directory, "snapshot.json")
        self.data, self.fsyncs, self._batching = {}, 0, False
        self.replayed, self.discarded = self._recover()
        self.log = open(self.log_path, "ab")

    def _append(self, op):
        payload = json.dumps(op).encode()
        self.log.write(self.HEADER.pack(len(payload), zlib.crc32(payload)) + payload)
        if not self._batching:
            self._sync()

    def _sync(self):
```

```

        self.log.flush(); os.fsync(self.log.fileno()); self.fsyzns += 1  # the commit point

@contextmanager
def batch(self):  # group commit: many records, one
    fsync
    self._batching = True
    try:
        yield
    finally:
        self._batching = False
        self._sync()

def put(self, key, value):
    self._append(["put", key, value]); self.data[key] = value  # log first, then apply
def delete(self, key):
    self._append(["del", key]); self.data.pop(key, None)
def get(self, key):
    return self.data.get(key)

def checkpoint(self):
    tmp = self.snap_path + ".tmp"
    with open(tmp, "w") as f:
        json.dump(self.data, f); f.flush(); os.fsync(f.fileno())
    os.replace(tmp, self.snap_path)  # the snapshot appears atomically
    self.log.close()
    open(self.log_path, "wb").close()  # everything before it is now
    redundant
    self.log = open(self.log_path, "ab")

def _recover(self):
    if os.path.exists(self.snap_path):
        with open(self.snap_path) as f:
            self.data = json.load(f)
    if not os.path.exists(self.log_path):
        return 0, 0
    with open(self.log_path, "rb") as f:
        buf = f.read()
    pos = replayed = 0
    while pos + self.HEADER.size <= len(buf):
        length, crc = self.HEADER.unpack_from(buf, pos)
        payload = buf[pos + self.HEADER.size: pos + self.HEADER.size + length]
        if len(payload) < length or zlib.crc32(payload) != crc:
            break  # torn or corrupt: stop at the last
            good record
        op = json.loads(payload)
        if op[0] == "put": self.data[op[1]] = op[2]
        else: self.data.pop(op[1], None)
        pos += self.HEADER.size + length
        replayed += 1
    discarded = len(buf) - pos
    if discarded:
        with open(self.log_path, "r+b") as f:
            f.truncate(pos)  # cut off the unusable tail
    return replayed, discarded

d = tempfile.mkdtemp()
kv = KVStore(d)
kv.put("alice", 100); kv.put("bob", 50); kv.put("carol", 75); kv.delete("carol")
before = dict(kv.data)
del kv  # crash: no shutdown, no close
kv = KVStore(d)
print(f"1. crash and restart: replayed {kv.replayed} records, state recovered: {kv.data ==
    before} -> {kv.data}")

kv.put("dave", 20)
with open(kv.log_path, "r+b") as f:  # as if the power failed mid-write,

```

```

before
f.truncate(os.path.getsize(kv.log_path) - 5)           # fsync returned: the record is
only partly there
kv = KVStore(d)
print(f"2. torn final write:  replayed {kv.replayed}, discarded {kv.discarded} bytes of the
partial record -> {kv.data}")

with open(kv.log_path, "r+b") as f:                   # a bit flips inside the second
record
f.seek(KVStore.HEADER.size + 30); byte = f.read(1); f.seek(-1, 1); f.write(bytes([byte[0] ^
0xFF]))
kv = KVStore(d)
print(f"3. corrupted record:  replayed {kv.replayed} before the checksum failed, discarded
{kv.discarded} bytes -> {kv.data}")

kv = KVStore(tempfile.mkdtemp())
for i in range(1000):
    kv.put(f"key{i}", i)
size_before = os.path.getsize(kv.log_path)
kv.checkpoint()
kv.put("after", 1); kv.put("checkpoint", 2)
kv = KVStore(os.path.dirname(kv.log_path))
print(f"4. checkpoint: log shrank from {size_before:,} bytes to 0; restart loaded the snapshot
"
      f"and replayed {kv.replayed} records -> {len(kv.data)} keys")

kv = KVStore(tempfile.mkdtemp())
for i in range(200):
    kv.put(f"a{i}", i)
one_each = kv.fsyncs
kv.fsyncs = 0
for chunk in range(4):
    with kv.batch():
        for i in range(50):
            kv.put(f"b{chunk}-{i}", i)
print(f"5. group commit: 200 writes, one fsync each = {one_each} fsyncs; in batches of 50 =
{kv.fsyncs} fsyncs")

```

OUTPUT

```

1. crash and restart: replayed 4 records, state recovered: True -> {'alice': 100, 'bob': 50}
2. torn final write:  replayed 4, discarded 22 bytes of the partial record -> {'alice': 100,
'bob': 50}
3. corrupted record:  replayed 1 before the checksum failed, discarded 78 bytes -> {'alice':
100}
4. checkpoint: log shrank from 29,780 bytes to 0; restart loaded the snapshot and replayed 2
records -> 1002 keys
5. group commit: 200 writes, one fsync each = 200 fsyncs; in batches of 50 = 4 fsyncs

```

Five ways to test a log

Crash and restart. The store is dropped without any shutdown, as if the process were killed. On restart it replays four records and reaches exactly the state it had acknowledged, with the delete of `carol` applied in order. Nothing in the dictionary itself was ever saved; the log was enough.

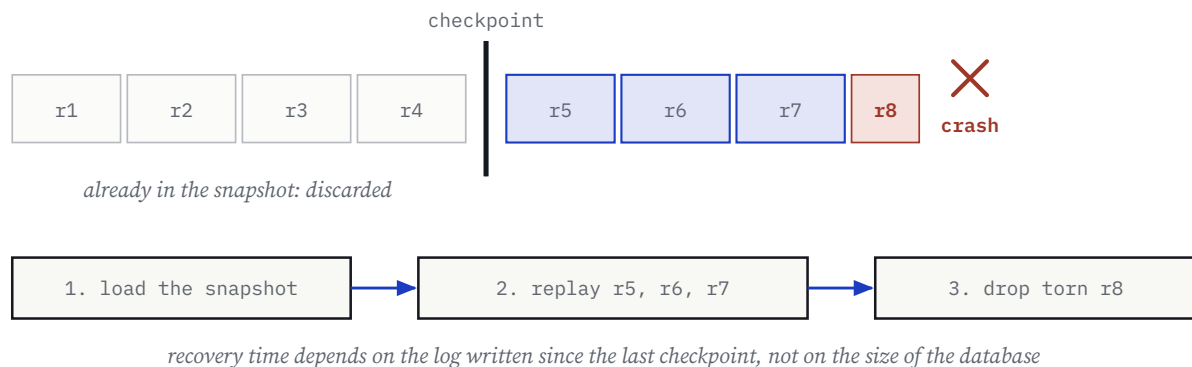
A torn final write. Power can fail while a record is being written, leaving only part of it on disk. Each record carries its length and a CRC-32 checksum of its contents, so recovery can tell a complete record from a fragment. It replays the four good records and discards the 22

bytes of the partial fifth. That's correct, not lossy: the fifth write never finished, so its `put` never returned, and no one was ever told it succeeded.

A corrupted record. A flipped bit in the second record fails its checksum, and recovery stops there, keeping only the first record and discarding everything after it. That looks harsh, since records three and four might be fine, but a log is only trustworthy as an unbroken prefix. Once one record is bad, recovery can't be sure where the next one starts, and replaying later changes without an earlier one could produce a state that never existed. PostgreSQL treats the first invalid record as the end of the log for the same reason. Real corruption in the middle of a log is a job for backups and replicas (chapter 16), not for recovery.

Checkpoints. A log that grows forever makes recovery slower every day. A **checkpoint** writes the current state durably (here, a snapshot file replaced atomically, the trick from chapter 1) and then discards the log before it. After a thousand writes, the log shrank from 29,780 bytes to nothing, and a restart loaded the snapshot and replayed just the two records written since. Real databases checkpoint by flushing dirty pages and noting the log position from which recovery must start, trading I/O spikes during checkpoints against shorter recovery.

Group commit. The flush is the expensive step: `fsync` takes anywhere from about a tenth of a millisecond on a good SSD to several milliseconds on a disk, and it caps how many commits per second one log can make. Batching commits that arrive together into one flush, called **group commit**, turned 200 flushes into 4 here. Databases do it automatically: when several transactions commit at once, one flush makes all of them durable.



Recovery after a crash. The checkpoint bounds how much log must be replayed. Records after it are redone in order, and the torn record at the end, whose write never completed and was never acknowledged, is detected by its checksum and dropped.

When `fsync` lies, and when pages tear

The whole scheme rests on `fsync` meaning what it says. Disks with volatile write caches can acknowledge a flush before the data is safe, which is why servers use drives with power-loss protection. The operating system can fail too. In 2018, PostgreSQL's developers

discovered that on Linux, a failed `fsync` could clear the error, so a retry would report success while the data was gone; PostgreSQL now stops with a panic on any `fsync` failure rather than retry. Data pages have their own torn-write problem. An 8 KB page written as two 4 KB halves can be half old and half new after a crash. PostgreSQL logs the full page image the first time a page changes after each checkpoint (`full_page_writes`), and InnoDB writes pages to a doublewrite buffer first.

The log is the database

The log turns out to be useful far beyond recovery. Ship it to another machine and replay it there, and you have a **replica** (chapter 16). Archive it next to a base backup, and you can restore the database to any moment, a minute before the bad `DELETE` : **point-in-time recovery**. Stream it to other systems, and every change becomes an event: **change data capture**, which is how tools like Debezium feed search indexes and warehouses. Seen this way, the log is the source of truth, and the tables are just a convenient, indexed copy of its latest state.

To finish the deliverable, swap the store's dictionary for chapter 6's B+tree, which adds ordered range scans, or put this log in front of chapter 7's memtable, which makes the LSM tree durable. Then test it the way this chapter did, but harder: kill the process at random moments thousands of times and check that every acknowledged write survives and no unacknowledged one appears.

COMMON TRAPS

Never testing recovery, so the first real crash is also the first test.

Retrying a failed `fsync` and believing the second answer.

Trusting consumer disks' write caches on a database server.

Deleting log files by hand to free space, which can make the database unrecoverable.

Checkpoints tuned badly: too rare means long recoveries, too frequent means constant I/O spikes.

EXERCISES

EXERCISE 15.1 A crash during the checkpoint

The checkpoint writes `snapshot.json.tmp`, renames it over `snapshot.json`, then truncates the log. What happens if the process dies (a) after writing the temporary file but before the rename, and (b) after the rename but before the truncate? Is the store still correct?

SOLUTION

- (a) The old snapshot and the full log are both intact, and the temporary file is ignored, so recovery is exactly as if the checkpoint never started.
- (b) The new snapshot already contains every change in the log, and the log is replayed on top of it. That's correct only because replaying is **idempotent** here: every record sets a key to an absolute value or deletes it, so applying it twice gives the same result. If records said "add 10", replaying an already-applied record would double it. Real databases avoid the problem by stamping each page with the log position of its last change and skipping records that the page already reflects.

EXERCISE 15.2 Why stop at the first bad record?

A colleague proposes that recovery skip a record with a bad checksum and carry on with the ones after it, to lose less data. What can go wrong?

SOLUTION

Three things. First, the length field of a corrupt record may itself be corrupt, so recovery can't reliably find where the next record starts; it might misread the middle of a record as a header. Second, later records may depend on the skipped one: a transaction's commit record without its changes, or an update to a row whose insert was skipped. Replaying them creates a state that never existed, which is worse than a clean loss, because nobody can tell it's wrong. Third, a bad record in the middle of the log means the disk or file system is damaged, so the right response is to stop, alert, and restore from a backup or replica rather than paper over it.

EXERCISE 15.3 Test the deliverable

Design an automated test that would give you confidence in your key-value store's durability.

SOLUTION

Run a workload in a child process that writes random puts and deletes, and records each operation in a separate journal only after the store acknowledges it. Kill the child with `SIGKILL` at a random moment,

restart the store, and check two properties: every acknowledged operation is reflected, and the state equals some prefix of the attempted operations, so nothing appears that wasn't written. Repeat thousands of times with different seeds. Add targeted cases: truncate the log at every byte offset of the last record, flip random bits, crash in each step of the checkpoint, and fill the disk. To simulate power loss rather than process death, run it on a file system you can drop unflushed writes from, such as a virtual machine snapshot or a fault-injecting file system.

CHECKLIST

- ☐ Can you state the write-ahead rule and why it makes commits cheap?
- ☐ Do your log records carry lengths and checksums, and does recovery stop at the first bad one?
- ☐ Is replay idempotent, or guarded by page log positions?
- ☐ Do checkpoints bound your recovery time?
- ☐ Have you tested recovery by killing the process thousands of times?

GO DEEPER

REFERENCE [Write-ahead logging PostgreSQL documentation](#)

ESSAY [The Log: what every software engineer should know Jay Kreps](#)

PAPER [ARIES: a transaction recovery method Mohan et al., 1992](#)

PART V

Running it

Two chapters on operating a database that people depend on. Replication, which copies the log to survive losing a machine, and backups, which survive what replication can't; then changing the schema of a table while the application keeps writing to it.

Replication, backups and recovery points

3 exercises

Replication copies the log to other machines, so you can survive losing one and spread reads across several. It is not a backup: a replica faithfully copies every mistake, including the `DELETE` with the wrong `WHERE`. Decide how much data you can afford to lose and how long you can afford to be down, then build replication for one and backups for the other.

Copying the log

Chapter 15 ended with the observation that the log is the database. Replication is that idea in practice: the primary streams its log to replicas, and each replica replays it to keep an identical copy. **Physical** replication ships the log's bytes and produces an exact copy of the primary's files, page for page. It's simple and complete, but the replica must run the same major version. **Logical** replication ships row changes, which can cross versions, replicate selected tables, and feed other systems, at the cost of more moving parts. PostgreSQL offers both; MySQL replicates its binary log, preferably in row format. The other choice is timing. With **asynchronous** replication, the primary acknowledges a commit without waiting for any replica. With **synchronous** replication it waits for at least one, which adds a network round trip to every commit. Semi-synchronous schemes wait for the replica to receive the record but not to apply it.

EXAMPLE replication.py · replica lag, failover and a point-in-time restore

```
class Primary:
    """Every change is a numbered log record (its LSN); replicas replay the same log."""
    def __init__(self):
        self.log, self.data = [], {}
    def write(self, key, value):
        self.log.append((key, value))
        self._apply(self.data, key, value)
        return len(self.log)  # the LSN of this change
    @staticmethod
    def _apply(data, key, value):
        if value is None: data.pop(key, None)
        else: data[key] = value

class Replica:
    def __init__(self, primary):
        self.primary, self.applied, self.data = primary, 0, {}
    def catch_up(self, records=None):
        end = len(self.primary.log) if records is None else min(len(self.primary.log),
            self.applied + records)
        for key, value in self.primary.log[self.applied:end]:
            Primary._apply(self.data, key, value)
        self.applied = end

p = Primary(); r1, r2 = Replica(p), Replica(p)
```

```

for i in range(100):
    p.write(f"user{i}", f"name {i}")
r1.catch_up(); r2.catch_up()

print("1. read your own writes")
lsn = p.write("user7", "Ana Lima") # the user saves a new display
name...
print("    read from a lagging replica:  ", r1.data["user7"], "<- the save looks lost")
def read(key, replica, token):
    """Serve from the replica only if it has replayed the client's last write."""
    return (replica.data[key], "replica") if replica.applied >= token else (p.data[key],
    "primary")
print("    with the client's LSN token:  ", read("user7", r1, lsn))

print("2. monotonic reads")
r1.catch_up() # r1 is now current, r2 still lags
print(f"    first read from r1: {r1.data['user7']!r}, next read from r2: {r2.data['user7']!r} <-
    time went backwards")

print("3. failover with asynchronous replication")
for i in range(5):
    p.write(f"order{i}", "paid") # acknowledged to customers
r1.catch_up(records=2) # then the primary's disk dies
lost = len(p.log) - r1.applied
print(f"    promoted replica is {lost} records behind: {lost} acknowledged orders are gone (RPO
    > 0)")
print("    synchronous replication would have made each commit wait for the replica: no loss,
    slower commits")

print("4. a replica is not a backup")
r1.catch_up(); r2.catch_up()
backup_lsn, backup = len(p.log), dict(p.data) # last night's base backup
p.write("user42", "renamed")
bad = p.write("user7", None) # someone runs DELETE with the wrong
    WHERE
r1.catch_up()
print("    replica after the DELETE:      ", r1.data.get("user7"), "<- replicated faithfully")
restored = dict(backup)
for key, value in p.log[backup_lsn:bad - 1]: # replay the archived log up to just
    before it
    Primary._apply(restored, key, value)
print("    point-in-time restore:          ", restored.get("user7"), "| later change kept:",
    restored.get("user42"))

```

OUTPUT

```

1. read your own writes
  read from a lagging replica:   name 7 <- the save looks lost
  with the client's LSN token:   ('Ana Lima', 'primary')

2. monotonic reads
  first read from r1: 'Ana Lima', next read from r2: 'name 7' <- time went backwards

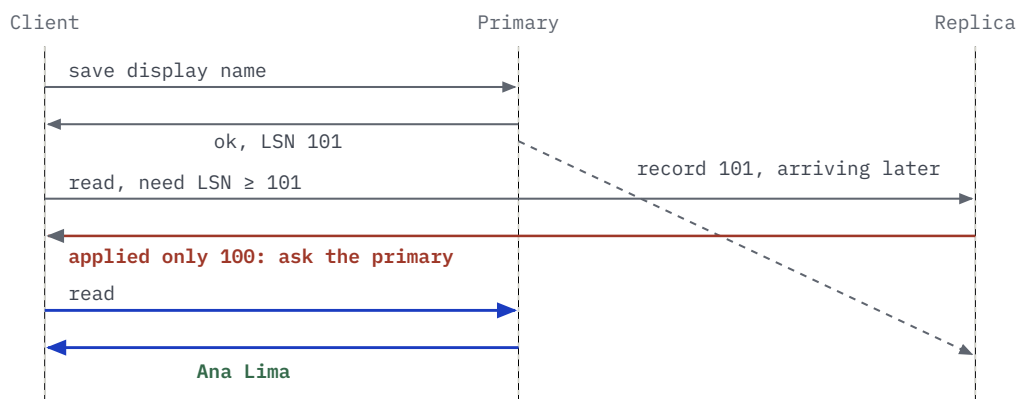
3. failover with asynchronous replication
  promoted replica is 3 records behind: 3 acknowledged orders are gone (RPO > 0)
  synchronous replication would have made each commit wait for the replica: no loss, slower
  commits

4. a replica is not a backup
  replica after the DELETE:      None <- replicated faithfully
  point-in-time restore:        Ana Lima | later change kept: renamed

```

What lag does to users

An asynchronous replica is always a little behind, usually milliseconds and sometimes much more. Part 1 shows the classic consequence: a user saves a new display name, the next page reads from a replica that hasn't replayed the change yet, and the save looks lost. The fix is **read-your-writes consistency**. The primary returns the log position (LSN) of each write, the client keeps it, and a read is served by a replica only if that replica has replayed at least that far; otherwise it goes to the primary, or waits. PostgreSQL exposes a replica's position with `pg_last_wal_replay_lsn()`, and MySQL uses global transaction ids for the same purpose. A cruder fix sends a user's reads to the primary for a few seconds after any write. Part 2 is the subtler **monotonic reads** problem: two replicas with different lag, and a user whose reads alternate between them sees data go forward and then back in time. Pinning each user to one replica, or using the same position tokens, prevents it.



Read-your-writes with a log position token. Without the token, the replica would have answered with the old name, and the user's save would look lost. With it, the replica knows it is too far behind for this client and the read goes to the primary.

Failover, and what it can lose

When the primary dies, a replica is promoted to take its place. With asynchronous replication, whatever the replica hadn't received is gone: in part 3, three orders that customers were told had succeeded. Two numbers make this a decision instead of a surprise. The **recovery point objective (RPO)** is how much data you can afford to lose, and the **recovery time objective (RTO)** is how long you can afford to be down. An RPO of zero requires synchronous replication to at least one other machine, and pays a round trip on every commit; a few seconds of RPO allows asynchronous. Failover also has to make sure the old primary is truly dead, or **fenced**, before the new one accepts writes, or both accept writes at once. In October 2018, a 43-second network partition led GitHub's failover tooling to promote primaries in a different region while the original side kept writing too. Reconciling the two sets of writes took 24 hours of degraded service.

Backups are for mistakes

Part 4 is the case replication can't help with. Someone runs a `DELETE` with the wrong condition, and the replica, doing its job, deletes the row too. The defence is **point-in-time recovery**: a base backup, plus every log record archived since, replayed to the moment just before the mistake. The restore brings back the deleted row and keeps the legitimate change made just before it. Backups must also live somewhere a mistake or an attacker can't reach at the same time: another account, another region. And they must be tested, because a backup you've never restored is a hope, not a backup. In January 2017, an engineer at GitLab, fighting a replication problem late at night, deleted the primary's data directory instead of the replica's. The team then found that their regular backups had been failing silently. They restored from a snapshot taken six hours earlier for another purpose, and lost six hours of users' issues, merge requests and comments.

COMMON TRAPS

Treating replicas as backups, when they copy every mistake within milliseconds.

Never restoring a backup until the day you need it.

Reading from replicas with no lag handling, so saves look lost and data goes back in time.

Asynchronous failover without an agreed RPO, so lost writes come as a surprise.

Failover without fencing, leaving two primaries accepting writes.

EXERCISES

EXERCISE 16.1 Set the objectives

Choose an RPO, an RTO and a matching setup for: (a) a payments ledger; (b) an analytics event store rebuilt daily from upstream logs; (c) the metadata for user-uploaded documents.

SOLUTION

- a. **RPO zero, RTO minutes.** Synchronous replication to at least one replica, preferably in another availability zone, with automated failover and fencing. The extra commit latency is the price of never losing an acknowledged payment. Add continuous log archiving for point-in-time recovery.
- b. **RPO of a day, RTO of hours.** The data can be regenerated, so daily backups, or none at all beyond the upstream logs, are enough. Spend the money elsewhere.

- c. **RPO of seconds, RTO under an hour.** Losing a few seconds of uploads is tolerable if users can retry; losing the metadata for files that exist is not. Asynchronous replication plus continuous log archiving, and a reconciliation job that compares metadata with the file store after any failover.

EXERCISE 16.2 The disappearing save

After users save their profile, they sometimes see the old values for about a second. Reads go to replicas. Give two fixes and their trade-offs.

SOLUTION

First, **log position tokens**, as in the example: return the write's LSN to the client (in a cookie or session), and serve the user's reads from a replica only once it has replayed that far. It's precise and keeps most reads on replicas, but every read path has to carry the token, and the routing layer must check replica positions. Second, **primary reads after writes**: for a few seconds after a user writes, send all of that user's reads to the primary. It's easy to implement, but puts extra load on the primary and guesses at a lag bound that a slow replica can exceed. A third, narrow fix: after a save, render the page from the values just written rather than re-reading them.

EXERCISE 16.3 A backup plan you can trust

Design the backup strategy for a 2 TB PostgreSQL database, including how you'll know it works.

SOLUTION

Take base backups regularly, say weekly, with a tool such as pgBackRest or the managed service's snapshots, and archive the log continuously, so you can restore to any moment within the retention window. Store both in a different account and region from the database, encrypted, with retention locks so a compromised production account can't delete them. Then prove it works: every month, an automated job restores the latest backup to a fresh instance, replays the log to a chosen time, runs verification queries (row counts, recent records, a checksum of a key table), and reports how long it took. That duration is your real RTO. Alert if the restore fails or the newest archived log is older than a few minutes.

CHECKLIST

- ☐ Have you agreed an RPO and an RTO for each important database?
- ☐ Does your replication mode match the RPO?
- ☐ Do replica reads handle lag where users would notice?
- ☐ Is the log archived continuously, to a separate account and region?
- ☐ Is a full restore tested automatically, on a schedule?

GO DEEPER

POST-MORTEM

Postmortem of the database outage of January 31 [GitLab](#)

POST-MORTEM

October 21 post-incident analysis [GitHub](#)

REFERENCE

Continuous archiving and point-in-time recovery [PostgreSQL documentation](#)

Changing the schema under live traffic

3 exercises

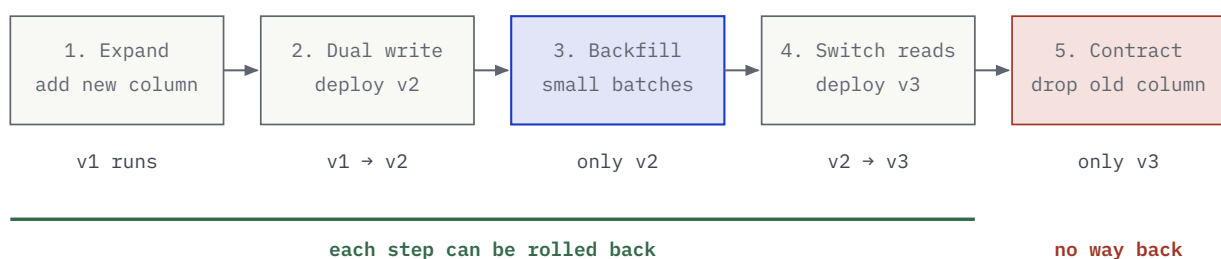
Every schema change on a live system happens while two versions of the application are running at once, and the database has to work for both. The safe pattern is expand and contract: add the new shape alongside the old one, move code and data across in small steps that can each be undone, and remove the old shape only when nothing uses it any more.

Why it's hard

On a laptop, renaming a column is one statement. In production, three things get in the way. The first is **two versions of the code**: a rolling deploy replaces servers one at a time, so for minutes or hours the old code and the new code run side by side against the same tables, and a rollback brings the old code back. Rename the column in one step and whichever version doesn't know the new name breaks. The second is **locks**. Many `ALTER TABLE` forms take an exclusive lock, and some rewrite the whole table while holding it. Even a change that needs the lock only for a moment must first wait for every running query on the table, and in PostgreSQL every query that arrives after it waits behind it, so a two-second change can stall the table for minutes behind one slow report. The third is **the data**: copying values into a new column across a hundred million rows takes time and can't be one transaction.

Expand and contract

The pattern breaks a change into steps, each compatible with both the code running before it and the code after it, and each reversible until the last.



the backfill starts only when no server still runs code that writes the old column alone

Expand and contract. The schema always supports every version of the code that might be running. The old column is dropped only after the code that reads it is gone, so the irreversible step comes last and carries no risk.

EXAMPLE

migration.py · renaming a column under live writes, in the wrong order and the right one

```
import random, sqlite3

def fresh_db():
    db = sqlite3.connect(":memory:", isolation_level=None)
    db.execute("CREATE TABLE users(id INTEGER PRIMARY KEY, fullname TEXT NOT NULL)")
    db.executemany("INSERT INTO users(fullname) VALUES (?)", ((f"User {i}",) for i in
        range(10_000)))
    return db

# The application versions that will be running at various times.
def v1_rename(db, uid, name):
    # the old code: knows only fullname
    db.execute("UPDATE users SET fullname = ? WHERE id = ?", (name, uid))
def v2_rename(db, uid, name):
    # the transitional code: writes both columns
    db.execute("UPDATE users SET fullname = ?, display_name = ? WHERE id = ?", (name, name,
        uid))

def backfill(db, live_writer, batch=1000):
    """Copy old values into the new column in short batches, with live traffic between
        batches."""
    largest = 0
    for start in range(1, 10_001, batch):
        db.execute("BEGIN")
        n = db.execute("UPDATE users SET display_name = fullname WHERE id BETWEEN ? AND ? "
            "AND display_name IS NULL", (start, start + batch - 1)).rowcount
        db.execute("COMMIT")
        largest = max(largest, n)
        for _ in range(50):
            # the application keeps working
            # meanwhile
            live_writer(db, random.randint(1, 10_000), f"Renamed {random.randint(1, 10**6)}")
    return largest

def mismatches(db):
    return db.execute("SELECT COUNT(*) FROM users WHERE display_name IS NOT
        fullname").fetchone()[0]

random.seed(17)
print("wrong order: backfill while v1 is still deployed")
db = fresh_db()
db.execute("ALTER TABLE users ADD COLUMN display_name TEXT")    # expand: instant, nothing
    breaks
backfill(db, v1_rename)
print(f"    rows where the new column is stale or missing after backfill: {mismatches(db)}")

print("right order: expand, deploy v2 everywhere, then backfill")
db = fresh_db()
db.execute("ALTER TABLE users ADD COLUMN display_name TEXT")    # 1. expand
largest = backfill(db, v2_rename)                                # 2-3. dual-write deployed,
    then backfill
print(f"    mismatches after backfill: {mismatches(db)}    largest single transaction:
    {largest:,} rows")
print("    4. verify: 0 mismatches, so v3 can read display_name only")
db.execute("ALTER TABLE users DROP COLUMN fullname")            # 5. contract, once v1 and v2
    are gone
print("    5. contract: columns now", [r[1] for r in db.execute("PRAGMA table_info(users)"]])
```

OUTPUT

```
wrong order: backfill while v1 is still deployed
    rows where the new column is stale or missing after backfill: 262
right order: expand, deploy v2 everywhere, then backfill
    mismatches after backfill: 0    largest single transaction: 1,000 rows
```

```
4. verify: 0 mismatches, so v3 can read display_name only
5. contract: columns now ['id', 'display_name']
```

The order is the whole trick

The first run backfills while the old code, v1, is still deployed. v1 knows nothing about `display_name`, so every rename it performs after a batch has passed that row updates `fullname` and leaves `display_name` holding the old value. The backfill finishes, everything looks done, and 262 rows are silently wrong, waiting to show users their old names the day v3 starts reading the new column. The second run deploys the dual-writing v2 everywhere first, so every rename keeps both columns in step, and only then backfills: zero mismatches. The batches matter too. Each one is its own short transaction of at most 1,000 rows, so no lock is held for long and replicas never fall far behind. A single `UPDATE` of the whole table would be one enormous transaction, holding locks and an old snapshot for its whole duration (chapter 14). Before switching reads, verify, as the example's mismatch count does, and only drop the old column once v1 and v2 are gone.

Engine features that help

Databases have steadily made more changes cheap. In PostgreSQL, adding a column with a constant default has been instant since version 11. `CREATE INDEX CONCURRENTLY` builds an index without blocking writes, and a constraint can be added `NOT VALID`, which checks new rows immediately, then validated against old rows later without a long lock. Setting `lock_timeout` before any DDL makes a change give up and retry rather than queue the whole table behind it. MySQL supports `ALGORITHM=INSTANT` for many column changes. For the rest, tools such as gh-ost and pt-online-schema-change build a shadow copy of the table, keep it in sync, and swap it in. SQLite supports adding, renaming and, since version 3.35, dropping columns. Whatever the engine, rehearse every migration on a copy of production-sized data before running it for real.

COMMON TRAPS

Renaming or dropping a column in one step while the old code may still be running or be rolled back to.

Backfilling before dual writes are fully deployed, leaving silently stale rows.

DDL without a lock timeout, so one long query turns a quick change into an outage.

Building indexes without the concurrent option on a busy table.

One giant backfill transaction instead of small batches.

EXERCISE 17.1 Make email unique

`users.email` should be unique, but the table already contains duplicates, and the sign-up form keeps writing. Plan the migration.

SOLUTION

First stop new duplicates at the source. Deploy code that normalises emails (lower case, trimmed) and checks before inserting, knowing the check can race (exercise 2.2). Then find the existing duplicates with `GROUP BY lower(email) HAVING count(*) > 1` and resolve them with the business: merge accounts, or rename the losers to a placeholder and notify them. Build the index without blocking writes: `CREATE UNIQUE INDEX CONCURRENTLY users_email ON users(lower(email))` in PostgreSQL. If new duplicates slip in during the build, it fails, leaving an invalid index to drop and retry. Once it succeeds, the database enforces the rule, and the application's check becomes a friendly early warning rather than the real protection.

EXERCISE 17.2 Which writes went stale?

In the wrong-order run, which writes produced the 262 stale rows, and why didn't renames by v1 *before* a row's batch cause problems?

SOLUTION

The stale rows are exactly those that v1 renamed *after* the backfill had already copied them. The batch set `display_name` from the then-current `fullname`, and v1's later update changed `fullname` without touching `display_name`. A v1 rename *before* the row's batch was harmless, because the batch copied the new `fullname` when it reached the row. So the danger window is any old-code write that lands after the copy, which is why dual writes have to be deployed everywhere before the backfill starts.

EXERCISE 17.3 The two-second outage that lasted three minutes

An `ALTER TABLE orders ADD COLUMN` that takes two seconds in testing caused three minutes of errors in production. A nightly report was running at the time. Explain, and prevent it next time.

SOLUTION

The `ALTER` needed an exclusive lock on `orders`, so it waited for the report's query, which held a shared lock. In PostgreSQL, the waiting `ALTER` sits in the lock queue ahead of every later request, so every ordinary read and write on `orders` queued behind it. The table was effectively locked until the report finished three minutes later and the `ALTER` ran. Prevent it by setting `SET lock_timeout = '2s'` before the DDL, so it gives up quickly instead of blocking everyone, and retrying with backoff until it gets the lock in a quiet moment. Schedule DDL away from long batch jobs, and check for long-running queries before starting.

CHECKLIST

- ☐ Is every migration compatible with both the current and the previous version of the code?
- ☐ Do backfills run in small batches, after dual writes are fully deployed?
- ☐ Do you verify the data before switching reads?
- ☐ Does all DDL run with a lock timeout and a retry?
- ☐ Have you rehearsed the migration on production-sized data?

GO DEEPER

CASE STUDY

Online migrations at scale Stripe engineering

TOOL

gh-ost: online schema migrations for MySQL GitHub

GUIDE

Strong Migrations: which changes are dangerous and why Andrew Kane

PART VI

Beyond the relational core

Three chapters on the data systems that sit beside, or instead of, a relational database. The NoSQL families and the access patterns each one makes cheap, the caches and search indexes that keep copies of the database's data, and the analytical stores that turn rows into columns.

NoSQL: choosing by access pattern

3 exercises

Non-relational databases aren't a newer, faster kind of relational database. Each family gives up something general, usually ad-hoc queries, joins or transactions across many rows, to make one access pattern cheap at large scale. Choose one when you know your access patterns and they fit its shape; choose relational when you don't, or when they'll change.

The families

Key-value stores (Redis, DynamoDB used by key) do one thing: get and put a value by its key, extremely fast. **Document** databases (MongoDB, Couchbase) store a nested JSON-like document per key and can index fields inside it. **Wide-column** stores (Cassandra, ScyllaDB, Bigtable, HBase) group rows into partitions by a partition key and keep the rows within a partition sorted by a clustering key, on LSM storage (chapter 7) that is built for writes. **Graph** databases (Neo4j, Amazon Neptune) store relationships as first-class links, so a query that follows them many hops deep doesn't become a pile of joins. **Time-series** databases (TimescaleDB, InfluxDB) are tuned for appending timestamped measurements and querying recent ranges. **Search engines** (Elasticsearch, OpenSearch) are the subject of the next chapter. What they share is a bargain: decide your queries up front, and the database makes those queries cheap at a scale where one relational server would struggle.

EXAMPLE nosql.py · one dataset, three models, three access patterns

```
import random, sqlite3

random.seed(18)
customers = range(1, 2001)
orders = []                                     # (order_id, customer_id, date)
items = []                                     # (order_id, product_id, qty)
for oid in range(1, 20_001):
    orders.append((oid, random.choice(customers), f"2026-{random.randint(1, 9):02d}-{
        random.randint(1, 28):02d}"))
    for product in random.sample(range(1, 501), random.randint(1, 4)):
        items.append((oid, product, random.randint(1, 3)))

# Model 1: relational, with indexes for the queries we know about.
db = sqlite3.connect(":memory:")
db.executescript("""
CREATE TABLE orders(id INTEGER PRIMARY KEY, customer_id INTEGER, created_at TEXT);
CREATE TABLE items(order_id INTEGER, product_id INTEGER, qty INTEGER);
CREATE INDEX orders_by_customer ON orders(customer_id, created_at);
CREATE INDEX items_by_order ON items(order_id);
CREATE INDEX items_by_product ON items(product_id);
""")
db.executemany("INSERT INTO orders VALUES (?, ?, ?)", orders)
db.executemany("INSERT INTO items VALUES (?, ?, ?)", items)
```

```

# Model 2: documents. One document per customer, with orders and items embedded.
docs = {c: {"customer": c, "orders": []} for c in customers}
by_id = {}
for oid, cid, date in orders:
    by_id[oid] = {"id": oid, "date": date, "items": []}
    docs[cid]["orders"].append(by_id[oid])
for oid, pid, qty in items:
    by_id[oid]["items"].append({"product": pid, "qty": qty})

# Model 3: one wide-column/key-value table, keys designed around the access patterns
# (the single-table style used with DynamoDB): partition key -> sorted items.
kv = {}
for oid, cid, date in orders:
    kv.setdefault(f"CUST#{cid}", {})[f"ORDER#{date}#{oid}"] = {"order": oid}
    kv.setdefault(f"ORDER#{oid}", {})[f"META"] = {"customer": cid, "date": date}
for oid, pid, qty in items:
    kv[f"ORDER#{oid}"][f"ITEM#{pid}"] = {"qty": qty}

cid, oid, product = 77, 4242, 123

def relational(sql, params):
    """Run the query, and describe its cost from SQLite's own plan."""
    plan = [r[-1] for r in db.execute("EXPLAIN QUERY PLAN " + sql, params)]
    searches = sum(line.startswith("SEARCH") for line in plan)
    scans = sum(line.startswith("SCAN") for line in plan)
    label = f"{searches} index search" + ("es" if searches > 1 else "") if not scans else f"{scans} table scan"
    return db.execute(sql, params).fetchall(), label

print(f'{"access pattern":>42}{relational:>18}{documents:>16}{key-value:>18}')

rows, a_rel = relational("SELECT id FROM orders WHERE customer_id = ? ORDER BY created_at DESC LIMIT 5", (cid,))
recent = sorted(docs[cid]["orders"], key=lambda o: o["date"], reverse=True)[:5] # read
# 1 document
part = sorted(kv[f"CUST#{cid}"].items(), reverse=True)[:5] # read
# 1 partition
print(f'{"A. a customer and their 5 latest orders":>42}{a_rel:>18}{1 document:>16}{1 partition:>18}')

order, b_rel = relational("""SELECT o.id, i.product_id FROM orders o JOIN items i ON i.order_id = o.id
                           WHERE o.id = ?""", (oid,))
found = [d for d in docs.values() if any(o["id"] == oid for o in d["orders"])] # no
# key leads to an order
print(f'{"B. one order with its items":>42}{b_rel:>18}{scan {len(docs):,>16}{1 partition:>18}')

(rel, c_rel = relational("SELECT COUNT(DISTINCT order_id) FROM items WHERE product_id = ?", (product,))
rel = rel[0]
doc_hits = sum(1 for d in docs.values() for o in d["orders"] if any(i["product"] == product for i in o["items"]))
kv_hits = sum(1 for pk, part_ in kv.items() if pk.startswith("ORDER#") and f"ITEM#{product}" in part_)
print(f'{"C. every order containing product 123":>42}{c_rel:>18}{scan {len(docs):,>16}{scan {len(kv):,>18}')
print(f'\nsame answers: A {[r[0] for r in rows] == [o["id"] for o in recent] == [v["order"] for _, v in part]}, "
      f"B {len(found) == 1 and len(order) == len(kv[f"ORDER#{oid}"]) - 1}, C {rel == doc_hits == kv_hits} ({rel} orders)')

```

OUTPUT

access pattern	relational	documents	key-value
A. a customer and their 5 latest orders	1 index search	1 document	1 partition

B. one order with its items	2 index searches	scan 2,000	1 partition
C. every order containing product 123	1 index search	scan 2,000	scan 22,000
same answers: A True, B True, C True (86 orders)			

Reading the table

The example models the same 20,000 orders three ways and asks three questions. The relational column comes from SQLite's own plans, and with the right indexes every answer is an index search. The document model, one document per customer with orders embedded, is perfect for question A: one read returns the customer and every order. The key-value model was designed for A and B: the customer's partition holds its orders sorted by date, and each order has a partition of its own.

Then the questions go off-script. Finding one order in the document model means scanning every customer document, because nothing leads from an order id to the customer who placed it; a real document database would need a secondary index on the embedded field, which is fine, as long as someone planned it. Question C, every order containing a product, is the one nobody planned. The relational database answers it with an index that can be added in a minute at any time. Both NoSQL models must scan everything, or have their data reorganised around the new question: a secondary index, or a copy of every order line keyed by product, kept in sync by every write. That is the trade in one table. NoSQL makes the anticipated access patterns cheap and the unanticipated ones expensive.

Designing for a NoSQL store

The method is the reverse of relational design. Start from the list of access patterns, not from the entities, and design keys so that each pattern is a lookup by key or a range within one partition. That is the single-table design style used with DynamoDB, where keys like `CUST#77` and `ORDER#2026-09-01#4242` encode the queries they serve. Denormalise on purpose and accept duplication. Watch the sizes: a partition or document that grows without bound, such as every comment on a viral post, becomes a hot spot or hits a hard limit (MongoDB documents are capped at 16 MB). And know what you're giving up. Joins move into application code, multi-record transactions are limited or missing (MongoDB and DynamoDB both added them later, with restrictions), and the schema is enforced, if at all, by your code reading the data.

When to choose one

NoSQL earns its place for enormous scale with simple, known patterns (Discord's message history, chapter 7), for records whose shape genuinely varies, for caching and sessions, for deep relationship traversals, and for high-rate time-series ingestion. Two developments have narrowed the gap. Relational databases absorbed the most useful NoSQL features: PostgreSQL's `jsonb`, with indexes inside documents, covers many document-store use cases. And distributed SQL databases such as Google Spanner and CockroachDB offer SQL

and transactions across many machines. A sound default for a new product is a relational database, adding a specialised store when a specific access pattern demands it and you can name which one.

COMMON TRAPS

Choosing NoSQL "for scale" years before having the scale, and paying in lost flexibility meanwhile.

Modelling relationally in a document database, with references everywhere and joins done in application code.

Unbounded documents or partitions that become hot spots or hit size limits.

Forgetting the unplanned question, the report or admin screen that nobody listed as an access pattern.

Assuming strong consistency where the store defaults to eventual consistency for reads.

EXERCISES

EXERCISE 18.1 Choose the store

Choose a store for each, and say why: (a) web sessions looked up by session id that expire after 30 minutes; (b) "friends of friends who like this band"; (c) metrics from 100,000 devices every ten seconds; (d) an online shop's orders, which finance also reports on.

SOLUTION

- a. **Key-value**, such as Redis with a TTL per key: one lookup by key, automatic expiry, no queries across sessions.
- b. **Graph**, if traversals like this are central to the product: two hops plus a filter is a natural graph query and an awkward pile of self-joins. If it's one feature among many, a relational database with recursive queries may be enough.
- c. **Time-series** (or a wide-column store keyed by device and time): 10,000 writes a second, always appended, queried by device and recent range, then downsampled.

- d. **Relational.** Orders need transactions, constraints and the unplanned questions finance will certainly ask, and their scale fits comfortably on one well-tuned server for a long time.

EXERCISE 18.2 Design the keys

In a single-table key-value store, design keys that serve "a customer's orders between two dates" and "an order by its id" with one key lookup or partition range each.

SOLUTION

Store each order twice, or once plus an index. Under partition key `CUST#<customer>`, use sort key `ORDER#<date>#<order id>`, so a customer's orders are sorted by date, and "between two dates" is a range query from `ORDER#2026-07-01` to `ORDER#2026-09-30~` within one partition. Under partition key `ORDER#<order id>`, store the order's details and its items, so "order by id" is one partition read. In DynamoDB, the second copy is usually a global secondary index rather than a hand-written duplicate, which the database keeps in sync, eventually consistently.

EXERCISE 18.3 The document that grew

Each blog post document embeds all of its comments. Popular posts now have documents of several megabytes, every new comment rewrites the whole document, and one viral post is approaching the 16 MB limit. Redesign it.

SOLUTION

Move comments out of the post. Either store each comment as its own document with a `post_id` and an index on `(post_id, created_at)`, or use the bucket pattern: documents holding up to a hundred comments each, keyed by post and bucket number, so pages of comments are one read and a new comment touches only the newest bucket. Keep a comment count and perhaps the three latest comments embedded in the post for the common case of rendering a post preview. The general rule: embed what is bounded and read together, and reference what grows without limit.

- ☐ Have you listed your access patterns before choosing a data store?
- ☐ Can you name the access pattern that justifies each non-relational store you run?
- ☐ Are documents and partitions bounded in size?
- ☐ Do you know how you'd answer an unplanned question against each store?
- ☐ Do you know each store's consistency guarantees for reads?

GO DEEPER

GUIDE The what, why and when of single-table design [Alex DeBrie](#)

REFERENCE Data modeling [MongoDB documentation](#)

BOOK Designing Data-Intensive Applications, chapter 2 [Martin Kleppmann](#)

Caches and search indexes beside the database

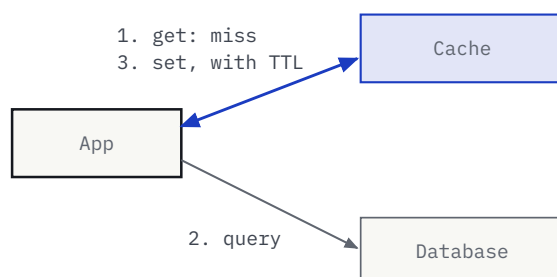
3 exercises

Two kinds of system sit beside almost every database: a cache that answers repeated questions from memory, and a search index that answers questions the database's indexes can't. Both are copies of data the database owns, which is what makes them fast and what makes them stale. Design them as derived data, with a clear answer to the question of what happens when the copy is wrong.

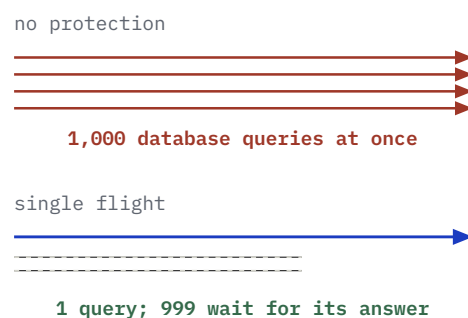
Cache-aside

The most common pattern is **cache-aside**, or lazy loading. To read, the application asks the cache first. On a hit it's done. On a miss it queries the database, stores the result in the cache with a time-to-live (TTL), and returns it. To write, it updates the database and deletes the cached entry, so the next read reloads it. Redis and Memcached are the usual caches. Alternatives exist: **write-through** updates the cache and the database together, and **write-behind** updates the cache and writes to the database later, which is fast and can lose data. Whatever the pattern, the TTL is the safety net: however the cache goes wrong, the error expires.

cache-aside



a popular key expires



Cache-aside, and what happens when a hot entry expires. Without protection, every request that misses goes to the database at the same moment. Single flight lets one request refresh the entry while the others wait for it, or are served the old value meanwhile.

EXAMPLE `cache.py` · a stampede, an invalidation race, and two kinds of search index

```
import sqlite3
```

```
# 1. A cache stampede: a popular key expires and 1,000 requests arrive in the same instant.
```

```

class Cache:
    def __init__(self): self.value, self.expired, self.refreshing = "v1", True, False

def serve(requests, strategy):
    cache, db_queries, waited = Cache(), 0, 0
    for _ in range(requests):
        if not cache.expired:
            continue # a normal hit
        if strategy == "no protection":
            db_queries += 1 # every miss queries the database
        elif strategy == "single flight":
            if cache.refreshing: waited += 1 # wait for the one query in flight
            else: cache.refreshing = True; db_queries += 1
        elif strategy == "serve stale, refresh once":
            if not cache.refreshing: cache.refreshing = True; db_queries += 1 # others get
            the old value now
    return db_queries, waited

print("1. stampede: 1,000 simultaneous requests for an expired key")
for strategy in ("no protection", "single flight", "serve stale, refresh once"):
    q, w = serve(1000, strategy)
    print(f"    {strategy:27} database queries: {q:5}    requests made to wait: {w}")

# 2. The invalidation race: read-through and write-invalidate interleaved badly.
db, cache = {"price": 100}, {}
log = []
stale_read = db["price"]; log.append("reader: cache miss, reads 100 from the database")
db["price"] = 120; log.append("writer: updates the database to 120")
cache.pop("price", None); log.append("writer: deletes the cache entry")
cache["price"] = stale_read; log.append("reader: writes the 100 it read into the cache")
print("\n2. invalidation race")
for line in log: print("    " + line)
print(f"    cache now says {cache['price']}, database says {db['price']}: stale until the entry
    expires")

# 3. Full-text search: an inverted index versus scanning every row.
docs = [
    "replica lag makes saves look lost", "the write ahead log makes commits durable",
    "a covering index avoids table lookups", "vacuum removes dead row versions",
    "the planner trusts its statistics", "a lagging replica can serve stale reads",
] * 2000
index = {}
for doc_id, text in enumerate(docs):
    for word in set(text.split()):
        index.setdefault(word, set()).add(doc_id) # word -> the documents containing
        it
hits = index["replica"] & index["stale"] # intersect two posting lists
print(f"\n3. search 'replica stale' over {len(docs):,} documents")
print(f"    inverted index: read {len(index['replica'])} + {len(index['stale'])},, postings,
    {len(hits):,} matches")
scan = [i for i, t in enumerate(docs) if "replica" in t and "stale" in t]
print(f"    LIKE-style scan: read {len(docs):,} documents, {len(scan):,} matches, same:
    {set(scan) == hits}")

fts = sqlite3.connect(":memory:")
fts.execute("CREATE VIRTUAL TABLE notes USING fts5(body)")
fts.executemany("INSERT INTO notes(body) VALUES (?)", [(d,) for d in docs[:6]])
print("    SQLite FTS5, ranked by bm25:",
    fts.execute("SELECT body FROM notes WHERE notes MATCH 'replica' ORDER BY
        bm25(notes)").fetchall())

```

OUTPUT

```

1. stampede: 1,000 simultaneous requests for an expired key
    no protection          database queries: 1000    requests made to wait: 0
    single flight          database queries:    1    requests made to wait: 999

```

```

serve stale, refresh once    database queries:    1    requests made to wait: 0

2. invalidation race
reader: cache miss, reads 100 from the database
writer: updates the database to 120
writer: deletes the cache entry
reader: writes the 100 it read into the cache
cache now says 100, database says 120: stale until the entry expires

3. search 'replica stale' over 12,000 documents
inverted index: read 6,000 postings, 2,000 matches
LIKE-style scan: read 12,000 documents, 2,000 matches, same: True
SQLite FTS5, ranked by bm25: [('replica lag makes saves look lost',), ('a lagging replica
    can serve stale reads',)]

```

The stampede

A cache can hide a database's limits so well that nobody notices until it fails. When a popular entry expires, every request that arrives before it's refilled misses at once, and in part 1 a thousand of them query the database at the same moment for the same answer. That's a **cache stampede**, also called a thundering herd, and it can take down a database that handles the steady load easily. Three fixes appear in the example and in practice. **Single flight** lets the first request refresh the entry while the others wait for its result: one query, 999 short waits. **Serve stale while revalidating** returns the old value to everyone and refreshes it once in the background: one query, nobody waits. **Early or jittered expiry** refreshes a hot entry slightly before its TTL ends, at a random moment, so entries don't all expire together. A restart has the same effect at a larger scale: an empty cache sends every request to the database, so warm the cache, or let traffic back in gradually.

The invalidation race

Part 2 shows why "delete the cache entry after writing" isn't the end of the story. A reader misses and reads 100 from the database. Before it stores that value, a writer updates the database to 120 and deletes the cache entry. Then the reader stores the 100 it read. The cache now contradicts the database, and will until the TTL expires. Every step was correct in isolation; the order made it wrong. Facebook's memcache paper describes the fix it adopted: **leases**. A miss hands the reader a token, a delete invalidates outstanding tokens, and the cache refuses a set with an invalidated token. Simpler defences are a short TTL, which bounds how long the staleness lasts, and **versioned keys** that include the row's version or update time, so a stale value is stored under a key nobody reads any more.

Search indexes

A B-tree finds rows whose value starts with something; it can't find documents that contain a word somewhere in the middle, so `LIKE '%stale%'` scans every row. Search needs an **inverted index**: for every word, the list of documents that contain it, called its postings. A query for two words intersects two posting lists. In part 3, the search read 6,000 postings to find its 2,000 matches, where the scan read all 12,000 documents; the gap grows with the size of the collection and the rarity of the words. Real search engines add tokenisation,

stemming (so *lagging* matches *lag*), and ranking, usually by BM25, which weighs rare words more heavily and rewards documents where the query's words are prominent. SQLite's FTS5, run at the end of the example, does exactly this inside the database.

Start with your database's built-in search, such as PostgreSQL's `tsvector` with a GIN index, SQLite's FTS5 or MySQL's `FULLTEXT`: it stays transactionally consistent with the data and needs nothing new to run. Move to a dedicated engine such as Elasticsearch or OpenSearch when you need typo tolerance, faceting, relevance tuning, or scale beyond one server. Feed it from the database's change stream (chapter 15) and accept that it lags slightly behind. Keep a way to rebuild it from scratch, because a derived index that drifts needs a reindex, and a reindex is only possible if the database is the source of truth.

COMMON TRAPS

Cache entries with no TTL, so any invalidation bug lasts forever.

Caching errors or empty results with the same TTL as real answers.

No stampede protection on the few keys that carry most of the traffic.

Treating the cache as the source of truth, so a cache restart loses data.

A search index with no rebuild path, drifting further from the database every week.

EXERCISES

EXERCISE 19.1 Choose a caching strategy

Choose what and how to cache: (a) a product page viewed a million times a day, whose price changes a few times a day; (b) each user's dashboard, built from three cheap indexed queries; (c) currency exchange rates fetched from a slow external API.

SOLUTION

- a. Cache the page data, shared across users, with cache-aside, a TTL of a few minutes, deletion on price change, and stampede protection, because a million daily views make it a hot key. Serving stale for a few seconds while revalidating is usually acceptable for a product page; check whether the price must be exact at checkout and read it from the database there.

- b. Probably don't cache it. Three cheap indexed queries per view are what the database is for, and per-user entries have low hit rates and add invalidation work. Cache it only if measurements show load.
- c. Cache aggressively, with a TTL that matches how fresh rates must be, and serve the stale value if the API is down, so a slow or failing dependency degrades the page instead of breaking it. A background job that refreshes rates every minute removes the miss path from user requests entirely.

EXERCISE 19.2 Close the race

Fix part 2's race using versioned keys, and explain why it works.

SOLUTION

Give the row a version that changes on every write (an integer, or the update timestamp), and include it in the cache key: `price:42:v7`. The reader learns the current version when it reads from the database, so it caches 100 under `price:42:v7`. The writer's update makes the version 8, and subsequent readers look up `price:42:v8`, miss, and load 120. The reader's late write of the stale 100 lands under the old `v7` key, which nobody reads any more and which expires on its own. The cost: finding the current version needs a cheap lookup of its own, or the version must travel with whatever references the item.

EXERCISE 19.3 Built-in or dedicated search?

Choose the database's built-in full-text search or a dedicated engine for: (a) 50,000 help-centre articles; (b) several terabytes of application logs a day; (c) a product catalogue needing typo tolerance, facets and tuned relevance.

SOLUTION

- a. **Built-in.** Fifty thousand articles is small; PostgreSQL's full-text search or SQLite's FTS5 handles it, stays consistent with the data, and adds no new system to run.
- b. **A dedicated engine** built for logs (OpenSearch, Elasticsearch, or a log-specific store). The volume is far beyond one database server, the data is append-only, and the queries are time-bounded searches.

c. **A dedicated engine.** Typo tolerance, faceted counts and relevance tuning are what search engines are built for, and doing them well inside a relational database is a project of its own. Feed it from the catalogue's change stream and keep a full reindex job.

CHECKLIST

- ☐ Does every cache entry have a TTL?
- ☐ Are hot keys protected from stampedes?
- ☐ Do you know how stale each cached value can be, and is that acceptable?
- ☐ Is the database, not the cache or the search index, the source of truth?
- ☐ Can you rebuild the search index from scratch?

GO DEEPER

PAPER [Scaling Memcache at Facebook](#) Nishtala et al., NSDI 2013

REFERENCE [Full text search PostgreSQL documentation](#)

REFERENCE [FTS5 SQLite documentation](#)

Analytical data: columns, stars and pipelines

3 exercises

Transactional databases are built to touch a few rows at a time; analysis touches millions of rows and only a few of their columns. Analytical systems flip the storage layout to keep each column together, which lets them read a fraction of the data and compress it dramatically, and they model data as facts surrounded by dimensions. Keep the two workloads apart, and move data between them deliberately.

Two workloads

An online shop's database spends its day on **transactional** (OLTP) work: show this order, add that item, update this stock count. Each query touches a handful of rows, by key, and needs an answer in milliseconds. The finance team's questions are **analytical** (OLAP): revenue by month and category for two years, the average basket size by region. Each query touches millions of rows but only a few columns, runs for seconds, and is asked by a few people. Run the second kind on a database built for the first and you get chapter 5's flooded cache and chapter 14's held-back vacuum, on the system your customers depend on. Analytical work belongs in a system built for it, fed from the transactional one.

Store columns, not rows

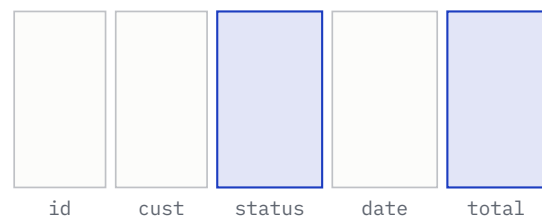
A row store keeps each row's values together in a page, which is ideal for fetching a whole row by key. A **column store** keeps each column's values together, so a query reads only the columns it mentions. Columns also compress far better than rows, because each holds one kind of value with lots of repetition.

row store: whole rows per page

17	·	4021	·	shipped	·	2026-09-14	·	2500
18	·	977	·	pending	·	2026-09-14	·	1200
19	·	4021	·	shipped	·	2026-09-15	·	880

every page read: 8.2 MB

column store: one column per file



two columns read: 3.0 MB

and each column compresses well, because it holds one kind of value

The same query against two layouts. In a row store, the two columns the query needs are interleaved with the three it doesn't, so it reads everything. In a column store, it reads two files and ignores the rest.

EXAMPLE `olap.py` · row versus column bytes, column compression, and a star schema rollup

```
import random, sqlite3, struct, zlib

random.seed(20)
N = 200_000
rows = [(i, random.randint(1, 50_000), random.choices(["shipped", "pending", "cancelled"], [85, 10, 5])[0],
             f"2026-{random.randint(1, 9):02d}-{random.randint(1, 28):02d}", random.randint(500, 50_000))
         for i in range(N)]

# 1. Row store versus column store for SELECT SUM(total) WHERE status = 'shipped'
row_bytes = sum(8 + 8 + len(s) + len(d) + 8 for _, _, s, d, _ in rows)  # every column of
                                every row
col = {"status": [r[2] for r in rows], "total": [r[4] for r in rows]}
col_bytes = sum(len(s) for s in col["status"]) + 8 * N  # only the two
                                columns used
print("1. SUM(total) WHERE status = 'shipped'")
print(f"   row store reads {row_bytes / 1e6:6.1f} MB (all five columns)")
print(f"   column store reads {col_bytes / 1e6:6.1f} MB (status and total only)")

# 2. Columns compress well because each holds one kind of value.
codes = {"shipped": 0, "pending": 1, "cancelled": 2}
status_dict = bytes(codes[s] for s in col["status"])  # dictionary encoding
srt = sorted(col["status"])
runs = sum(1 for i in range(N) if i == 0 or srt[i] != srt[i - 1])  # run-length after
                                sorting
dates = sorted(r[3] for r in rows)
day_numbers = [int(d[5:7]) * 31 + int(d[8:]) for d in dates]
deltas = bytes(min(255, day_numbers[i] - day_numbers[i - 1]) if i else 0 for i in range(N))  #
                                delta encoding
raw_status = sum(len(s) for s in col["status"])
print("2. compression of single columns")
print(f"   status: {raw_status:}, bytes as text -> {len(status_dict):}, as 1-byte codes -> "
      f"{runs} runs once sorted ({runs * 5} bytes)")
print(f"   dates: {sum(len(d) for d in dates):}, bytes as text -> "
      f"{len(zlib.compress(deltas)):}, bytes "
      f"as sorted day-to-day deltas, compressed")

# 3. A star schema: one fact table of measurements, small dimension tables of context.
db = sqlite3.connect(":memory:")
db.executescript("""
CREATE TABLE dim_date(date_key INTEGER PRIMARY KEY, day TEXT, month TEXT);
CREATE TABLE dim_product(product_key INTEGER PRIMARY KEY, name TEXT, category TEXT);
CREATE TABLE fact_sales(date_key INTEGER, product_key INTEGER, qty INTEGER, revenue_cents
                           INTEGER);
""")
days = [f"2026-{m:02d}-{d:02d}" for m in range(7, 10) for d in range(1, 29)]
db.executemany("INSERT INTO dim_date VALUES (?, ?, ?)", [(i, d, d[:7]) for i, d in
    enumerate(days)])
db.executemany("INSERT INTO dim_product VALUES (?, ?, ?)",
    [(p, f"product {p}", ["books", "games", "tools"][p % 3]) for p in range(30)])
db.executemany("INSERT INTO fact_sales VALUES (?, ?, ?, ?)",
    [(random.randrange(len(days)), random.randrange(30), q := random.randint(1, 5),
      q * random.randint(500, 3000))
     for _ in range(50_000)])
print("3. revenue by month and category, with subtotals (a ROLLUP, written out for SQLite)")
for month, category, revenue in db.execute("""
SELECT d.month, p.category, SUM(f.revenue_cents) FROM fact_sales f
JOIN dim_date d ON d.date_key = f.date_key JOIN dim_product p ON p.product_key =
f.product_key
GROUP BY d.month, p.category
UNION ALL
SELECT d.month, '(all)', SUM(f.revenue_cents) FROM fact_sales f
```

```
JOIN dim_date d ON d.date_key = f.date_key GROUP BY d.month
ORDER BY 1, 2""):
print(f"    {month}    {category:6} {revenue / 100:>12,.2f}")
```

OUTPUT

- SUM(total) WHERE status = 'shipped'
row store reads 8.2 MB (all five columns)
column store reads 3.0 MB (status and total only)
- compression of single columns
status: 1,419,838 bytes as text -> 200,000 as 1-byte codes -> 3 runs once sorted (15 bytes)
dates: 2,000,000 bytes as text -> 790 bytes as sorted day-to-day deltas, compressed
- revenue by month and category, with subtotals (a ROLLUP, written out for SQLite)

2026-07	(all)	871,563.76
2026-07	books	295,811.23
2026-07	games	297,398.29
2026-07	tools	278,354.24
2026-08	(all)	871,569.62
2026-08	books	291,120.00
2026-08	games	290,521.41
2026-08	tools	289,928.21
2026-09	(all)	875,846.43
2026-09	books	293,836.14
2026-09	games	297,582.99
2026-09	tools	284,427.30

Reading the numbers

For a sum of one column filtered by another, the row store reads all five columns of every row, 8.2 MB, while the column store reads two, 3.0 MB. That's before compression, which is where columns pull far ahead. The status column, 1.4 MB of repeated words, shrinks to 200,000 one-byte codes with **dictionary encoding**, and to three runs, 15 bytes, with **run-length encoding** once the table is sorted by status. Dates sorted in order become small day-to-day differences, which **delta encoding** plus a general compressor squeezes from 2 MB to under 800 bytes. A table can only be sorted one way, so column stores let you choose a sort key, and the leading columns of that key compress best. Engines then process compressed values in batches of thousands at a time (vectorised execution), which is how DuckDB on a laptop, or ClickHouse, BigQuery, Snowflake and Redshift at scale, scan billions of rows in seconds. Parquet files bring the same layout to data that lives in object storage.

Facts and dimensions

Analytical schemas are usually **star schemas**, the design Ralph Kimball popularised. A central **fact table** records events at a chosen **grain**, such as one row per order line, with numeric measures like quantity and revenue, plus keys to the **dimension tables** around it: date, product, customer, store. Dimensions are wide and denormalised on purpose, with month and quarter stored on the date row and category stored on the product row, because analytical data is loaded in bulk and read far more than it's changed, so chapter 2's arguments for normalisation weigh much less. History matters here: when a product changes category, a **type 2 slowly changing dimension** adds a new dimension row with validity dates rather than overwriting, so last year's sales stay in last year's category, the same idea as chapter 3's history tables.

The example's third part runs a typical star-schema query on SQLite: revenue by month and category, joined from the fact table to two dimensions, with a subtotal row per month. Databases that support it write the subtotals with `GROUP BY ROLLUP (month, category)`, or more generally `GROUPING SETS` and `CUBE`, which compute several groupings in one pass; MySQL offers `WITH ROLLUP`. SQLite has none of these, so the example writes the rollup out as a `UNION ALL` of two groupings.

Pipelines

Data reaches the analytical store through a pipeline. Classic **ETL** extracts from the source, transforms in a separate system, and loads the result. Modern **ELT** loads raw data into the warehouse first and transforms it there with SQL, often managed with dbt, which keeps the raw data available for new questions and makes transformations versioned and testable. Batch loads run on a schedule; streaming loads follow the source's change log (chapter 15) for fresher data. Whichever you choose, treat the pipeline like production code: loads that are **idempotent**, so a rerun doesn't double-count; tests for row counts, nulls and uniqueness at each step; and a stated freshness target that someone is alerted about when it's missed.

COMMON TRAPS

Running analytics on the production primary, competing with customers for cache and I/O.

An unclear grain, so some rows mean an order and others an order line, and every sum is wrong.

Overwriting dimension history when reports need to see the past as it was.

Pipelines that aren't idempotent, double-counting a day's data whenever a job is retried.

Using a column store for point lookups and frequent updates, the workload it's worst at.

EXERCISES

EXERCISE 20.1 Where the savings come from

The column store read 3.0 MB against the row store's 8.2 MB. Estimate what it would read with dictionary-encoded status and a sort key starting with status, and explain the other ways the engine can skip work.

SOLUTION

Dictionary encoding cuts status from about 1.4 MB to 0.2 MB, and sorting by status reduces it to a few runs, a few bytes. The total column stays around 1.6 MB of 8-byte integers, and less with bit-packing, since the values fit in 16 bits, so the whole query reads roughly 0.4 to 1.6 MB instead of 3.0. With status sorted, the engine also skips work: per-block minimum and maximum values (zone maps) tell it which blocks of `total` belong to `'shipped'` rows, so it reads only those blocks. And it can sum the compressed values in vectorised batches without materialising rows. Twenty to a hundred times less I/O than the row store is typical for queries like this.

EXERCISE 20.2 Design a star

Design a star schema for a ride-hailing company's trips: state the grain, the measures and the dimensions, and say which dimension needs history.

SOLUTION

Grain: one row per completed or cancelled trip. Measures: fare, distance, duration, wait time, tip, and a cancellation flag. Dimensions: date and time of request (with hour, weekday, holiday flags), rider, driver, pickup and drop-off zone, vehicle type, and payment method. The driver dimension needs history: drivers change vehicle, city and rating tier, and a report on last quarter should attribute trips to the driver's attributes at the time. Make it a type 2 slowly changing dimension with valid-from and valid-to dates, and have each trip reference the driver row that was current when it happened. Keep the grain strict: per-trip promotions or ratings, if several can apply to one trip, belong in separate fact tables at their own grain.

EXERCISE 20.3 ETL or ELT?

A startup has its product data in PostgreSQL and wants dashboards in a cloud warehouse. Recommend ETL or ELT and outline the pipeline.

SOLUTION

ELT. Replicate the source tables into the warehouse as they are, using a managed connector or change data capture from PostgreSQL's logical replication, so analytics never queries the production database. Transform inside the warehouse with SQL models in dbt: staging models that clean and rename, then fact and dimension models at clear grains. dbt's tests check uniqueness, non-null keys and referential integrity on every run. Keep the raw replicated tables, since new

questions often need fields nobody modelled. ELT suits a startup because the warehouse's compute does the heavy lifting, the transformations live in version control, and the team needs SQL rather than a separate transformation platform.

CHECKLIST

- ☐ Does analytical work run somewhere other than the transactional primary?
- ☐ Do you know why columnar storage reads less and compresses better?
- ☐ Does every fact table have a stated grain?
- ☐ Do dimensions that change keep the history reports need?
- ☐ Are your pipelines idempotent, tested, and held to a freshness target?

GO DEEPER

BOOK The Data Warehouse Toolkit Ralph Kimball & Margy Ross

SURVEY The design and implementation of modern column-oriented database systems Abadi et al., 2013

REFERENCE Why DuckDB DuckDB

One per chapter, each linked back to it. Read them before you design a schema, and against every change that touches a query, an index, a transaction or a migration.

-
- 01 Know which promises you need: writes that are atomic and durable, concurrency that loses nothing, and finding without scanning. A file keeps none of them.
-
- 02 Declare every rule your code assumes as a constraint, test that it is enforced, and store each fact in one place.
-
- 03 Design trees, money, history and polymorphism on purpose: integer cents, UTC timestamps and real foreign keys.
-
- 04 Count the queries each page sends, load what loops touch eagerly, and write SQL where the ORM fights the relational model.
-
- 05 Estimate cost in pages, keep the working set in memory, and keep big scans away from the primary's cache.
-
- 06 Keep keys short and sequential, so B-trees stay shallow, dense and cache-friendly.
-
- 07 Choose B-trees for predictable reads and LSM trees for heavy writes, and know which amplification you are paying.
-
- 08 Put equality columns before the range column, keep indexed columns bare, and drop the indexes nobody uses.
-
- 09 Read plans as operator trees, spot the blocking operators, and index the columns you join and sort on.
-
- 10 When a plan is bad, compare estimated rows with actual rows and fix the statistics before reaching for hints.
-
- 11 Tune by experiment: realistic data, one change at a time, median timings, identical results, and every plan written down.
-
- 12 Put each business operation in one short transaction, and never call

another service from inside it.

-
- 13 Know your isolation level and the anomalies it allows; make read-modify-write atomic and retry serialization failures.
-
- 14 Watch for long transactions, lock rows in a consistent order, and use version checks for edits that span human time.
-
- 15 Log before you apply, checksum every record, stop recovery at the first bad one, and test recovery by crashing on purpose.
-
- 16 Agree an RPO and an RTO; replicate for one, back up for the other, and restore a backup on a schedule.
-
- 17 Change schemas by expand and contract, backfill in small batches after dual writes are deployed, and run DDL with a lock timeout.
-
- 18 Choose a non-relational store only for an access pattern you can name, and plan for the question nobody listed.
-
- 19 Give every cached value a TTL and every hot key stampede protection, and keep the database as the source of truth.
-
- 20 Keep analytics off the transactional primary: columns for scans, stars with a clear grain, and pipelines you can rerun safely.
-

READING LIST

The courses and books behind this volume, and the chapters each one deepens. If you read only one book, make it *Designing Data-Intensive Applications*. If you take only one course, take CMU 15-445: its projects build a buffer pool, a B+tree index and a query executor, the same parts this volume builds in miniature.

WORK	AUTHOR	DEEPENS
SELECT Again (interactive course)	this series' companion SQL course	the SQL itself
CMU 15-445/645 Database Systems (course)	Andy Pavlo	Ch. 5–15
Designing Data-Intensive Applications	Martin Kleppmann	Ch. 5–7, 12–20
Database Internals	Alex Petrov	Ch. 5–7, 15
Use The Index, Luke	Markus Winand	Ch. 6, 8–11
Architecture of a Database System	Hellerstein, Stonebraker & Hamilton	Ch. 1, 9–10, 12–15
SQL Antipatterns	Bill Karwin	Ch. 2–4
The Data Warehouse Toolkit	Ralph Kimball & Margy Ross	Ch. 20

Compiled October 2026 as the companion to stage 08, Databases and data modelling, of The Engineer's Syllabus, alongside *The Engineer's Craft*, *Shipyard*, *Fleet*, *Watch*, *Citadel*, *Foundry* and *Exchange*, and as the second half of a pair with *SELECT Again*, which teaches the SQL. Research drew on Hellerstein, Stonebraker and Hamilton's architecture survey, CMU 15-445, the SQLite documentation on atomic commit, query planning and FTS5, the PostgreSQL documentation on MVCC, write-ahead logging, planner statistics and point-in-time recovery, Use The Index, Luke, Berenson et al. on isolation levels, O'Neil et al. on LSM trees, Leis et al. on query optimisers, Jepsen, Facebook's memcache paper, Stripe on online migrations, and the public post-mortems and write-ups from GitLab, GitHub, Sentry, Uber and Discord. Engine versions and defaults are as of late 2026; check your engine's documentation before relying on one.

Every example was executed before publication with Python 3.14 and the SQLite 3.53 that ships with it, using only the standard library, and the output printed under each example is what it printed. A script then re-ran every example exactly as it appears on the page and compared its output with the page, character for character. The one exception is chapter 11's tuning session, whose timings vary from run to run: its output is one recorded run, and the re-run checks that it completes, returns identical results at every step and finishes under fifty milliseconds. Re-run again in October 2026 with Python 3.14.7 and SQLite 3.53, with the same result. Tested here: torn writes, atomic renames and lost updates; constraint enforcement and the foreign-key pragma trap; functional dependencies and keys; adjacency lists, closure tables and money; the N+1 query problem; a byte-level slotted page and a scan-resistant buffer pool; a B+tree with splits and range scans; an LSM tree with compaction and Bloom filters; SQLite's planner on composite, functional and covering indexes; four join algorithms and the iterator model; misestimates from skew and correlation; a million-row query tuned to under fifty milliseconds; transactions, savepoints and durability; three isolation levels across three anomalies; MVCC visibility, vacuum and deadlock detection; a key-value store with a checksummed write-ahead log, checkpoints and group commit; replica lag, failover and point-in-time recovery; an expand-and-contract migration; three data models across three access patterns; cache stampedes, an invalidation race, inverted indexes and FTS5; and columnar storage, compression and a star-schema rollup.