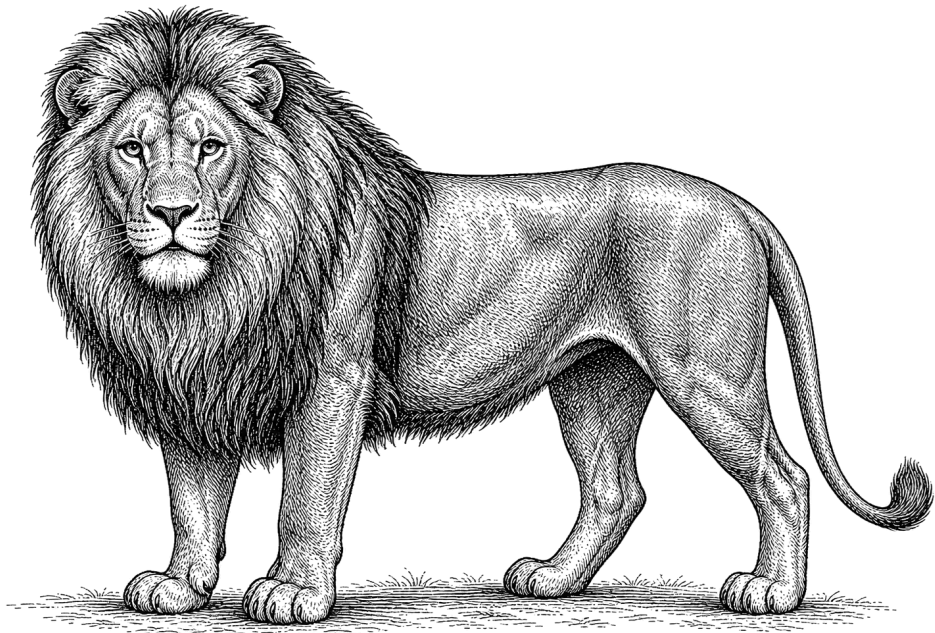




Database Engine-Level Thinking

Storage, memory, and the log underneath the query

PostgreSQL • DuckDB • Neo4j

Kareem Ashraf

Contents

Introduction	1
I PostgreSQL Internals	9
1 Storage and Memory Internals	12
1.1 Heap File Layout	12
1.1.1 What It Is	12
1.1.2 Why This Matters	15
1.1.3 Hands-On Inspection	17
1.1.4 Summary	17
1.2 TOAST: The Oversized-Attribute Storage Technique	18
1.2.1 What It Is	19
1.2.2 Why This Matters	20
1.2.3 Hands-On Inspection	22
1.2.4 Summary	23
1.3 The Visibility Map and Free Space Map	24
1.3.1 What They Are	24
1.3.2 Why Vacuum Depends on Both	25
1.3.3 Monitoring	30
1.3.4 Summary	31
1.4 The Buffer Pool and Clock-Sweep Eviction	32
1.4.1 What It Is	32

1.4.2	Why This Matters	34
1.4.3	Monitoring	36
1.4.4	Summary	36
1.5	WAL Internals	38
1.5.1	What It Is	38
1.5.2	Why This Matters	43
1.5.3	Monitoring	46
1.5.4	Summary	46
1.6	Full-Page Writes	47
1.6.1	What It Is	48
1.6.2	Why This Matters	50
1.6.3	Monitoring	52
1.6.4	Summary	52
1.7	Chapter Summary	54
2	Indexing Deep Dive	57
2.1	B-Tree and B+ Tree	57
2.1.1	What It Is	58
2.1.2	Why This Matters	62
2.1.3	Summary	66
2.2	Index-Only Scans and Covering Indexes	67
2.2.1	What It Is	67
2.2.2	Why This Matters	69
2.2.3	Hands-On Inspection	77
2.2.4	Summary	78
2.3	Hash Indexes	79
2.3.1	What It Is	79
2.3.2	Why This Matters	84

2.3.3	Hands-On Inspection	85
2.3.4	Summary	85
2.4	BRIN Indexes	86
2.4.1	What It Is	86
2.4.2	Why This Matters	91
2.4.3	Hands-On Inspection	92
2.4.4	Summary	92
2.5	GIN Indexes	93
2.5.1	What It Is	93
2.5.2	Why This Matters	98
2.5.3	Hands-On Inspection	100
2.5.4	Summary	100
2.6	GiST Indexes	101
2.6.1	What It Is	101
2.6.2	Why This Matters	106
2.6.3	Hands-On Inspection	106
2.6.4	Summary	108
2.7	Choosing an Index	109
2.7.1	What It Is	109
2.7.2	Why This Matters	112
2.7.3	Summary	113
2.8	Partial Indexes	114
2.8.1	What It Is	114
2.8.2	Why This Matters	115
2.8.3	Hands-On Inspection	118
2.8.4	Summary	120
2.9	Composite Indexes	120
2.9.1	What It Is	121

2.9.2	Why This Matters	121
2.9.3	Hands-On Inspection	127
2.9.4	Summary	128
2.10	Expression Indexes	129
2.10.1	What It Is	130
2.10.2	Why This Matters	130
2.10.3	Hands-On Inspection	134
2.10.4	Summary	135
2.11	Chapter Summary	136
3	MVCC and Concurrency	139
3.1	Why Versions, Not Locks	139
3.1.1	What It Is	140
3.1.2	Why This Matters	143
3.1.3	Summary	144
3.2	Tuple Headers	145
3.2.1	What It Is	146
3.2.2	Why This Matters	150
3.2.3	Hands-On Inspection	152
3.2.4	Summary	153
3.3	Snapshots	154
3.3.1	What It Is	154
3.3.2	Why This Matters	157
3.3.3	Monitoring	160
3.3.4	Summary	161
3.4	Visibility Rules	162
3.4.1	What It Is	162
3.4.2	Why This Matters	166

3.4.3	Summary	169
3.5	UPDATE and DELETE as Versioning	170
3.5.1	What It Is	171
3.5.2	Why This Matters	174
3.5.3	Summary	176
3.6	Chapter Summary	177

Introduction

This book is not a tutorial from an expert looking back. It is written alongside the learning — the questions I ask, what I think I have understood, and the same material rephrased into a reference I can return to. The aim is to stop only using databases and start understanding them.

About the Author



Kareem Ashraf

Kareem Ashraf

Backend software engineer

I started caring about databases for a concrete reason: I worked on a service that acted as a query builder, and the hard part was not writing SQL — it was making it fast. Indexing and vacuum helped, until they did not. I hit a ceiling where the usual knobs were not enough. To go further I had to understand the internals and the trade-offs behind them.

So I studied — books, articles, tutorials, and hands-on notes from running and breaking real systems. This book is that journey written down: what I have learned so far, what I am still learning, and how the internals connect to optimisation that shows up in production. Not trivia for its own sake — mechanisms with a use case, for backend engineers who are tired of guessing.

[LinkedIn](#) • [GitHub](#)

Why I Do This

I do this because I was tired of being competent on the surface and blind underneath.

I could write the SQL. I could add the index. I could tune the obvious knob when something went wrong. By the usual measures of a backend engineer, that looked like progress. Then production would do something my mental model could not explain — a query that was fast for a year suddenly was not, a table that lost most of its rows stayed the same size on disk, a replica drifted for no reason I could name, WAL filled the disk while write traffic looked flat, an index sat unused though it was shaped exactly for the query — and I would do what most of us do. Search the symptom. Apply a fix. Move on.

That path works. It just does not build anything. Each incident leaves you with one narrow fact and no structure to hang the next incident on. You collect remedies. You do not build a model. Without a model you cannot predict; you can only react. I did not want that to be the shape of my career.

So I chose a different loop: learn the mechanism, ask until the picture is honest, check what I think I understood, and write it down carefully enough that future-me — and anyone else on the same road — can use it as a reference. Not as a claim that I already know. As proof that I am willing to go deep where it matters.

I do this because databases are where I put that depth. Backend work has many places you can specialise; I picked the layer that holds state, because that is usually the hard part of scaling, and because guessing there is expensive. I do this in public enough that the writing has to stay clear, and humble enough that the writing has to stay honest.

That is why this book exists. Not to publish a finished expertise. To practise becoming the engineer I am trying to be — T-shaped, backend, vertical stroke in databases — one mechanism at a time.

What T-Shaped Actually Means Here

The T-shaped metaphor gets used loosely, and it is often misunderstood in exactly the way that made me write this section carefully.

When I say I am trying to become a T-shaped engineer, I do **not** mean a generalist who dabbles in everything. I mean a **backend engineer**. Backend work is already a wide field. Inside it you can put your depth in networking, in operating systems, in DevOps and infrastructure, in algorithms and data structures, in distributed systems, in security — any one of those can be a career. You then keep **breadth** across the others so you can talk to specialists, choose tools without ignorance, and not become useless the moment the problem sits one layer away from your specialty.

That is the T I am talking about: breadth across backend work, and one vertical stroke driven deep.

I chose the vertical stroke to be **databases**.

Not because servers do not matter — they do — but because a poor database will stop a system that healthy servers cannot save. Application servers scale horizontally by adding machines and vertically by giving them more CPU and RAM. That is still work, but it is comparatively mechanical. Databases are where state lives. They are harder to shard, harder to replicate correctly, harder to tune under write load, and far more expensive to get wrong. When an application feels slow, or fails to scale, or wakes someone at three in the morning, the cause is more often the data layer than the compute layer.

So the hard scaling problem — the one that decides whether the product survives contact with real traffic — is usually the database, not the server fleet. That is why the depth goes here.

The horizontal stroke, then, is breadth across engines and shapes of data: knowing that graph databases exist and roughly when to reach for one, knowing that columnar engines beat row stores on analytical scans, knowing the vocabulary well enough to have a productive conversation with someone who specialises in a system you have never run. Breadth keeps you from solving every problem with PostgreSQL because it was the only tool you knew.

The vertical stroke is depth inside that choice, and depth is different in kind, not just in degree. Depth is knowing why a mechanism was designed the way it was, what constraint forced that design, and what the design costs you. Depth is being

able to reason from first principles about a situation you have never encountered, and being right.

My horizontal stroke across engines was already fine. I had worked with PostgreSQL as my primary system, reached for DuckDB when analytical workloads called for it, and used Neo4j when relationships mattered more than rows. I could hold a conversation about any of them.

My vertical stroke — the database internals themselves — was thinner than I wanted to admit. I knew PostgreSQL's surface extremely well and its interior barely at all. I knew that `VACUUM` was important without knowing what it touched. I knew `shared_buffers` mattered without knowing what lived inside it or how pages left it. I knew WAL guaranteed durability without knowing what was written, when, or why it sometimes doubled in volume for no visible reason.

This book is the deliberate thickening of that vertical stroke: backend engineer, depth in databases, because that is where the system either scales or does not. I am not claiming to have arrived. I am documenting the path while I walk it.

Why Internals, and Why Now

There is a common objection to studying internals, and it deserves a direct answer: *isn't this a waste of time? The database abstracts all of it away. That is the entire point of a database.*

The abstraction is real and it is excellent. PostgreSQL will serve you well for years without you knowing what a line pointer is. But abstractions leak under load, and they leak in exactly the places where the cost of not understanding is highest — at three in the morning, under pressure, with a system degrading in a way that has no obvious cause.

More importantly, the leaks are not random. They are direct, traceable consequences of specific design decisions, and every one of those decisions is knowable.

Consider a concrete case. You have a table with an `id`, a `created_at`, and a large `jsonb` column. A query selecting only `id` and `created_at` is fast. A colleague adds the `jsonb` column to the select list for debugging, and the query becomes fifty times slower. Nothing else changed — same rows, same index, same plan shape.

Without a model, this is inexplicable and you start guessing. With a model, it is not even surprising. Large values do not live in the main table; they are sliced into chunks and stored elsewhere, and touching the column forces a separate lookup per row. The mechanism has a name, a threshold, and a configurable strategy. Once you know it exists, you never lose an afternoon to it again — and more valuably, you start designing schemas that avoid the problem before it happens.

That is the compounding effect that symptom-fixing never gives you. Every mechanism you genuinely understand makes the next unexplained behaviour easier to reason about, because these mechanisms are not independent. They interlock. Bloat, vacuum, index-only scans, checkpoint behaviour, and replication lag are not five topics. They are five views of one system.

What This Book Is

These pages are written **while I am learning**, not after I have finished.

I read, I ask questions, I check what I think I have understood, and then I rephrase it in the clearest form I can — first as a reference for my future self, and second so someone else on the same path might use it too. That is the whole method. It is not a collection of war stories from a long career, and it is not me presenting myself as an authority. I am a backend engineer trying to become T-shaped with depth in databases, and this book is the notebook of that attempt, rewritten carefully enough to be useful.

Each topic takes one mechanism and does three things. First, it explains what the mechanism actually is — the data structures, the algorithms, the file layouts, at a level of detail that lets you reason about behaviour rather than memorise rules. Second, it explains why that design exists: what constraint the engineers were working against, and what the design costs. Third, it connects the mechanism to things you can observe in a real system, with the queries and tools that let you see it directly.

The chapters are deliberately self-contained. You should be able to open the one on TOAST without having read the one on heap layout, because that is how reference material is actually used — you arrive with a question, not at the beginning. Where one mechanism genuinely depends on another, the chapter says so and points you to it.

Some conventions worth knowing before you start. Version-specific behaviour is noted explicitly, because PostgreSQL changes and a fact that is true today may have been false three major versions ago. Every configuration parameter comes with the reasoning behind its recommended value rather than just the value, because the number is worthless without the model that produced it. And where there is a common piece of advice I think is wrong or dangerously incomplete, I say so and explain why — as a learner testing the advice, not as a final verdict from on high.

What This Book Is Not

It is not an introduction to SQL or to relational databases. It assumes you can write a non-trivial query, know what an index is for, and have deployed something that people depend on.

It is not a replacement for the official documentation. The PostgreSQL manual is outstanding, and I am not attempting to duplicate it. Documentation tells you what a parameter does. This book tries to tell you what is happening underneath such that the parameter needs to exist at all.

It is not finished. It is a study plan executed in public — or privately for now — chapter by chapter, written as I learn rather than as a retrospective from someone who already knows. It will stay incomplete for a long time. That is the intended state. A reference that grows as its author does is more honest than one that pretends to completeness or expertise it has not earned.

How the Book Is Organised

The material is divided into five parts, ordered by dependency rather than difficulty.

Part	Focus
PostgreSQL Internals	Storage layout, memory management, write-ahead logging, query execution, indexing, MVCC, vacuum, and replication. The foundation everything else builds on.
Advanced SQL Mastery	SQL that is correct under concurrency and at scale. Every topic has a real failure mode or a non-obvious behaviour that surprises people in production.
DuckDB	Columnar storage, vectorised execution, and the analytical engine design that contrasts instructively with PostgreSQL's row-oriented model.
Neo4j	Native graph storage, traversal performance, and the algorithms that make graph databases worth reaching for.
Cross-Cutting Internals	Storage engine theory, distributed systems fundamentals, data formats, and the observability tooling that applies to every system above.

Part I comes first for a reason. The concepts in it — pages, tuples, visibility, buffering, logging — are the vocabulary every other part depends on. Understanding why PostgreSQL stores rows the way it does is what makes DuckDB's columnar design legible as a set of deliberate trade-offs rather than an arbitrary alternative.

How to Read It

If you are working through the material systematically, read Part I in order. The chapters are sequenced so that each one builds on the vocabulary of the ones before it, and the payoff comes at the end of the storage section, where the individual mechanisms resolve into a single coherent picture of what happens when a row is written.

If you arrived with a specific problem, go directly to the relevant chapter. Each one opens with what the mechanism is and closes with how to observe it, so you can get an answer without reading the surrounding material.

Either way, the most valuable thing you can do is run the queries. Every observability section contains SQL you can execute against a real database. Reading that the visibility map enables index-only scans is worth something. Watching `idx_tup_fetch` stay flat while `idx_tup_read` climbs, on your own data, is worth considerably more. The mechanisms in this book are not abstractions — they are files on a disk and counters in a catalog, and you can look at all of them.

Key point

The premise of this book in one sentence. Every performance problem you will encounter — bloat, slow vacuums, replication lag, checkpoint storms, unexplained I/O spikes, plans that ignore your indexes — traces back to a mechanism that is documented, observable, and learnable. The difference between guessing and knowing is not talent. It is whether you took the time to build the model.

PART I

PostgreSQL Internals

The foundation everything else builds on. Before query plans, concurrency tricks, or replication make sense, you need a picture of where bytes live on disk, how they reach memory, and how a crash is survived. This part builds that picture for PostgreSQL.

Chapters in this part

Ch.	Subject
-----	---------

1	Storage and Memory Internals — heap pages, TOAST, visibility and free-space maps, the buffer pool, WAL, and full-page writes. The six mechanisms that interlock when a row is written.
---	---

2	Indexing Deep Dive — B-tree shape, leaf splits, HOT updates, index-only scans, covering indexes, hash equality, BRIN summaries, GIN inverted lists, GiST predicate trees, choosing among them, partial, composite, and expression indexes.
---	---

3	MVCC and Concurrency — why versions instead of locks, tuple headers and <code>pg_xact</code> , snapshots and the cleanup horizon, visibility rules across scan types, and UPDATE/DELETE as versioning (with HOT as the MVCC optimisation). Locking, isolation deep dives, and SSI follow later in the chapter.
---	---

Ch.	Planned
-----	---------

4	Query execution — parsing, planning, and the executor
---	---

5	Vacuum and autovacuum — bloat, freeze, and what autovacuum actually does
---	--

6	Replication and high availability — streaming, slots, and failover
---	--

7	Extensions — how <code>CREATE EXTENSION</code> loads code into the server
---	---

8	Performance engineering — tying observability back to mechanisms
---	--

What you should be able to do after Part I

- Read `EXPLAIN (ANALYZE, BUFFERS)` output and connect buffer hits and I/O to the buffer pool and heap layout.
- Explain why an `UPDATE` creates bloat, why `VACUUM` matters, and why an index-only scan sometimes stops being index-only.
- Trace a write from SQL through WAL, the buffer pool, and a checkpoint — and name which component does each job.
- Take any heap tuple and say whether it is visible to a given snapshot, and name what holds the cleanup horizon back.
- Tune `shared_buffers`, checkpoint settings, and `wal_compression` with the model in mind, not by rote.

Key point

Start with Chapter 1 and read the topics in order the first time through. Each topic ends with its own summary; each chapter ends with a synthesising summary. Return to either summary when you need a quick refresh.

Storage and Memory Internals

Where the bytes actually live. How a row is laid out on a page, what happens when one will not fit, how PostgreSQL knows which pages it can skip, how any of it reaches memory, and how a crash is survived by writing the log first. Every later chapter assumes this one.

1.1 Heap File Layout

Pages, tuples, and line pointers — the physical foundation that makes MVCC, bloat, and index performance explicable rather than mysterious.

1.1.1 What It Is

When you create a table in PostgreSQL, the engine creates a file on disk — called a **heap file** — to store all the rows of that table. The word “heap” here does not imply sorted order; it simply means an unordered collection of row data. This file grows as you insert more data, and it is divided into fixed-size units called **pages** (also called **blocks**).

Each page is exactly **8 kilobytes** by default, and this is the smallest unit PostgreSQL ever reads or writes from disk. Even if you need just one byte from a row, PostgreSQL will load the entire 8 KB page that contains it into memory.

Every page has a precise, rigid internal structure. Understanding it is like reading the blueprint of a building — once you know where everything goes, nothing about PostgreSQL’s behaviour will surprise you again.

Anatomy of a Page

Here is what the inside of a single 8 KB page looks like, from top to bottom.

Page Header (24 bytes)
– LSN (last WAL position for this page)
– Checksum
– Flags (all-visible, all-frozen, etc.)
– pd_lower (end of line pointer array)
– pd_upper (start of tuple data)
– pd_special (for index pages)
Line Pointer Array (ItemId entries)
Each entry: 4 bytes
Stores: offset, length, and status flag
Array grows DOWNWARD as tuples added
Free Space (between pd_lower and pd_upper)
Tuple Data
Each tuple: header + null bitmap
+ optional OID
+ actual column values
Data grows UPWARD from end of page
Special Space (used only by index pages)

The Page Header

The page header is always exactly 24 bytes. It contains the LSN (the last write-ahead log position that affected this page), a checksum if enabled, and two critical offsets: `pd_lower`, which marks the end of the line pointer array, and `pd_upper`, which marks where tuple data begins.

Free space is everything between those two offsets. When they meet, the page is full.

Line Pointers

Line pointers are tiny 4-byte entries stored in the array just below the header. Each one contains a byte offset into the page where a particular tuple lives, the length of that tuple, and a status flag indicating whether the pointer is unused, in use, or pointing to a redirect.

Line pointers never move once created. When a tuple is deleted, its line pointer is marked dead, but the pointer slot stays in the array permanently until vacuum reclaims it. This is the foundation of PostgreSQL's MVCC model — old versions of rows are kept in place and dead pointers linger until cleanup.

The ctid: A Row's Physical Address

The physical address of any row in PostgreSQL is called its `ctid` (**current tuple identifier**), stored as `(page_number, line_pointer_index)`. Running `SELECT ctid, * FROM your_table` shows the raw physical location of each row.

Key point

When you update a row, PostgreSQL does **not** modify the existing tuple in place. It writes a brand new version of the row somewhere else in the heap (possibly on the same page if there is room, or on a new page), marks the old tuple as dead by setting its `t_xmax` field, and updates the index to point to the new `ctid`. This is why `ctid` changes after an `UPDATE`.

The Tuple Header

Each tuple's header (`HeapTupleHeaderData`) is 23 bytes long and carries the MVCC bookkeeping data.

Field	Meaning
<code>t_xmin</code>	Transaction ID that inserted this tuple
<code>t_xmax</code>	Transaction ID that deleted or updated this tuple (zero if still live)
<code>t_cid</code>	Command ID within the transaction
<code>t_ctid</code>	Pointer to the current version of this row (for update chains)
<code>infomask</code>	Flags: null columns, frozen state, xmin/xmax commit status, and more

After the header comes an optional null bitmap (one bit per column, present only if any column is null), and then the actual column values laid out consecutively according to their type's alignment requirements.

1.1.2 Why This Matters

Most PostgreSQL behaviour that seems mysterious at the surface becomes completely logical once you understand the heap page layout.

MVCC Becomes Tangible

PostgreSQL's multi-version concurrency control — the reason readers never block writers and writers never block readers — is implemented entirely inside this **tuple header**. Two transactions can look at the same page and see completely different tuples, because each transaction evaluates `t_xmin` and `t_xmax` against its own snapshot.

If you do not understand the **tuple header**, MVCC is magic. If you do understand it, MVCC is just bookkeeping.

Table Bloat Stops Being Mysterious

Every `UPDATE` leaves behind a **dead tuple**. Every `DELETE` leaves behind a dead tuple. Those dead tuples sit inside 8 KB pages, occupying space.

The pages those dead tuples live on cannot be returned to the operating system until vacuum reclaims the space and, in some cases, until an aggressive vacuum can truncate trailing empty pages from the heap file.

Trap

A table that was 10 GB, had 9 GB of rows deleted, and then had vacuum run on it can *still* be 10 GB on disk. The pages are now full of free space that PostgreSQL can reuse, but the file has not shrunk unless `VACUUM FULL` or `pg_repack` was used. Understanding the physical layout makes this intuitive rather than confusing.

Index Correlation Becomes Predictable

Because tuples are stored in insertion order and indexes store physical `ctid` references, if your rows are scattered randomly across the heap — for example after many updates — an index scan that finds 10,000 rows might need to read 10,000 different pages. That is brutal random I/O.

A `CLUSTER` operation rewrites the heap in index order, making index scans sequential. You cannot reason about this trade-off without knowing what the heap layout looks like.

1.1.3 Hands-On Inspection

The `pageinspect` contrib extension lets you read raw page contents in SQL. This is not a curiosity — it is the right tool when diagnosing corruption, unexpected bloat, or unusual vacuum behaviour.

```
CREATE EXTENSION IF NOT EXISTS pageinspect;

-- Read the page header of page 0
SELECT * FROM page_header(get_raw_page('mytable', 0));

-- Read all tuples on page 0, including xmin/xmax
SELECT lp, lp_off, lp_len, t_xmin, t_xmax, t_ctid
FROM heap_page_items(get_raw_page('mytable', 0));

-- See a row's physical address
SELECT ctid, * FROM your_table LIMIT 10;

-- Watch ctid change after an UPDATE
SELECT ctid, id FROM your_table WHERE id = 1;
UPDATE your_table SET updated_at = now() WHERE id = 1;
SELECT ctid, id FROM your_table WHERE id = 1;
```

1.1.4 Summary

At a glance

The page is the unit of I/O.

Every table is a heap file divided into 8 KB pages. PostgreSQL never reads less than a full page, so a single byte costs the same I/O as a whole page.

A page has four regions.

A 24-byte header, a line pointer array growing downward, free space in the middle, and tuple data growing upward. When `pd_lower` meets `pd_upper`, the page is full.

Line pointers are the indirection layer.

Indexes point at a `ctid` — a (page, line pointer) pair — not at a byte offset. That indirection is what lets tuples move within a page without touching indexes.

Updates never modify in place.

An UPDATE writes a new tuple elsewhere and marks the old one dead via `t_xmax`. This is why `ctid` changes after an update, and why updates create bloat.

MVCC lives in the tuple header.

`t_xmin` and `t_xmax` compared against a transaction's snapshot are the whole mechanism. Readers never block writers because each transaction evaluates visibility itself.

Deleting rows does not shrink the file.

Dead tuples leave reusable free space inside pages, but the file keeps its size unless `VACUUM FULL` or `pg_repack` rewrites it.

Physical order drives index cost.

Because indexes store physical addresses, scattered rows turn one index scan into thousands of random page reads. `CLUSTER` rewrites the heap in index order to fix this.

Use `pageinspect` to see it directly.

`page_header()` and `heap_page_items()` expose the real bytes when diagnosing bloat, corruption, or strange vacuum behaviour.

1.2 TOAST: The Oversized-Attribute Storage Technique

How PostgreSQL stores values that cannot possibly fit in a single page — and why the mechanism stays invisible until the afternoon it costs you fifty times your query latency.

1.2.1 What It Is

PostgreSQL has a hard constraint: **every row, in its entirety, must fit within a single 8 KB heap page**. A row cannot span multiple pages.

Given that columns can hold text documents, JSON blobs, binary data, or XML that easily runs into megabytes, something has to give. That something is **TOAST** — The Oversized-Attribute Storage Technique.

TOAST is a completely transparent mechanism. You never call it explicitly, and no application code changes are required. PostgreSQL activates it automatically when a row's on-disk size would exceed a certain threshold, and deactivates it transparently when you read the data back. From the application's perspective, the column is just there, full-sized, ready to use.

Two Mechanisms: Compression and Out-of-Line Storage

Compression uses the LZ family of algorithms (historically pglz, with LZ4 configurable since PostgreSQL 14 via `default_toast_compression`). If a large value can be compressed enough to fit inline, PostgreSQL compresses it and stores it directly in the main heap tuple. The tuple header's infomask bits record that the value is stored in compressed form.

Out-of-line storage splits the value into chunks of approximately 2000 bytes each and stores them in a separate table — the **TOAST table** — which lives in the `pg_toast` schema and is named `pg_toast_<oid>`, where `<oid>` is the OID of the parent table. Each chunk is a separate row in this TOAST table, identified by a chunk sequence number.

The main heap tuple stores a small **TOAST pointer** (18 bytes) in place of the actual value — a reference containing the OID of the TOAST table and the OID of the toasted value.

Key point

The threshold at which TOAST engages is `TOAST_TUPLE_THRESHOLD`, one quarter of a page — roughly **2000 bytes**. Once a row's total size would exceed this, PostgreSQL applies TOAST strategies to variable-length columns until the row fits, trying columns in order of size, largest first.

Storage Strategies

Each column's storage behaviour is governed by its **storage strategy**, which you can inspect and change.

Strategy	Behaviour
PLAIN	Never compress, never move out-of-line. Used for small or fixed-layout types.
EXTENDED	Compress first; if still too large, move out-of-line. Default for text, bytea, jsonb, and arrays.
EXTERNAL	Move out-of-line without compression. Faster substring access at the cost of disk space.
MAIN	Compress and try to keep inline; out-of-line only as a last resort.

```
-- Inspect current strategies for variable-length columns
SELECT attname, attstorage
FROM pg_attribute
WHERE attrelid = 'your_table'::regclass AND attlen = -1;

-- Change a column's strategy
ALTER TABLE documents ALTER COLUMN body SET STORAGE EXTERNAL;
```

1.2.2 Why This Matters

TOAST is one of those mechanisms that is invisible until it causes you serious pain. Engineers who do not know it exists get blindsided by behaviours they cannot explain.

Hidden I/O on “Simple” Queries

Suppose you have a table with three columns: `id bigint`, `metadata jsonb` (large, and therefore TOASTed), and `created_at timestampz`.

You run this query:

```
SELECT id, created_at
FROM events
ORDER BY created_at DESC
LIMIT 100;
```

This query does not touch the `metadata` column. PostgreSQL does not detoast it, there is no TOAST table I/O, and the query is fast.

Now a colleague adds `metadata` to the select list for debugging and commits it to production.

Trap

The query now reads a TOAST table for *every single row* in the result, each fetch potentially a separate **random I/O**. Query time can increase by 10 to 100 times. Nothing else changed — same rows, same index, same plan shape. Selecting a toasted column forces **detoasting**, and detoasting means TOAST table reads.

Index Limitations

You cannot directly index a toasted value beyond an index entry size limit — roughly one third of a page, or about 2700 bytes. Values exceeding this will either raise an error or require expression indexing strategies.

Text you need to search should be stored as a `tsvector` or indexed via a GIN index rather than a B-tree.

`pg_column_size()` Does Not Mean What You Think

`pg_column_size(value)` returns the size of the value *as stored in the column* — which may be the compressed size if the value is compressed inline, or just 18 bytes if it is a TOAST pointer. The actual data is much larger.

This regularly trips up engineers trying to estimate storage requirements from column sizes.

Vacuum Must Handle TOAST Tables Too

Dead TOAST chunks from updated or deleted rows must be vacuumed separately. The TOAST table has its own entry in `pg_stat_user_tables`.

If you see a TOAST table accumulating dead tuples, either your autovacuum configuration needs adjustment for that table specifically, or something is preventing TOAST vacuuming — long-running transactions hold back dead tuple removal everywhere, including here.

1.2.3 Hands-On Inspection

```
-- Find the TOAST table for a given table
SELECT relname, reltoastrelid::regclass AS toast_table
FROM pg_class
WHERE relname = 'your_table';

-- Query the TOAST table directly
SELECT chunk_id, chunk_seq, length(chunk_data) AS chunk_bytes
FROM pg_toast.pg_toast_16384
LIMIT 20;

-- Compare stored size against logical size
SELECT pg_column_size(body) AS stored_bytes,
       length(body)         AS logical_bytes
FROM documents
LIMIT 5;

-- Check dead tuples on TOAST tables specifically
SELECT relname, n_dead_tup, last_autovacuum
FROM pg_stat_all_tables
WHERE schemaname = 'pg_toast'
ORDER BY n_dead_tup DESC
```

```
LIMIT 10;
```

1.2.4 Summary

At a glance

A row must fit in one 8 KB page.

TOAST exists solely to satisfy that constraint. It is automatic, transparent, and always on — you never call it.

The threshold is roughly 2000 bytes.

At one quarter of a page, PostgreSQL starts applying TOAST to variable-length columns, largest first, until the row fits.

Two mechanisms, in order.

Compress the value to fit inline; if that is not enough, move it out of line into a separate TOAST table in chunks of about 2000 bytes, leaving an 18-byte pointer in the heap tuple.

Storage strategy is per column.

EXTENDED (compress, then move) is the default. EXTERNAL skips compression for faster substring access, MAIN prefers staying inline, PLAIN does neither.

Selecting a toasted column is the hidden cost.

Adding one large column to a select list forces detoasting for every row, each potentially a separate random read. Latency can rise ten- to a hundredfold with no plan change.

pg_column_size() can mislead.

It reports the stored size — possibly compressed, possibly just the 18-byte pointer — not the size of the actual data.

Large values resist B-tree indexing.

Index entries are capped near 2700 bytes. Searchable text belongs in a tsvector with a GIN index.

TOAST tables need vacuuming too.

They accumulate dead chunks and have their own statistics rows. Watch them separately in pg_stat_all_tables under the pg_toast schema.

1.3 The Visibility Map and Free Space Map

Two small fork files that decide whether autovacuum is cheap or ruinous, and whether your index-only scans are actually index-only.

1.3.1 What They Are

Every table in PostgreSQL has not just one heap file but potentially several **fork files** stored alongside it. Two of the most important are the **Visibility Map** and the **Free Space Map**.

These are not optional features you configure. They are always present, always maintained by PostgreSQL, and critical to both correctness and performance.

The Visibility Map

The VM is stored in a file named `<relfilenode>_vm`. It maintains **two bits per heap page**.

Bit	Meaning
all-visible	Every tuple on the page is visible to all current and future transactions. No dead tuples exist, and no in-progress transaction needs an older version.
all-frozen	Every tuple on the page has been <i>frozen</i> — its <code>t_xmin</code> replaced with <code>FrozenTransactionId</code> (the value 2), making it unconditionally visible to all transactions, past and future.

The all-visible bit is cleared the moment any tuple on the page is inserted, updated, or deleted. The all-frozen bit persists until the page is modified.

Frozen tuples are PostgreSQL's protection against **transaction ID wraparound** — the 32-bit transaction counter cycling back to zero and causing old tuples to appear to be in the future.

The Free Space Map

The FSM is stored in a file named `<relfilenode>_fsm`. It is a **binary tree of page-level free space summaries**.

At the leaves of the tree, each entry represents one heap page and stores an approximation of how much free space is available on that page. It is stored as a single byte, so the granularity is roughly 32 bytes.

Parent nodes store the maximum free space of all their children, so PostgreSQL can find a page with enough room for a new tuple in $O(\log N)$ time by traversing from root to leaf.

Vacuum updates the FSM after processing each heap page. `INSERT` and `UPDATE` operations consult it to find an appropriate page rather than scanning all pages linearly.

1.3.2 Why Vacuum Depends on Both

Vacuum is the process that reclaims space from dead tuples. But its relationship with the VM and FSM is deeper than “clean up dead rows” — it is a carefully orchestrated system where all three components work together.

The VM Makes Index-Only Scans Possible

Normally, when an index scan finds a matching entry, PostgreSQL must visit the heap page to check tuple visibility — confirming the row is actually visible to the current transaction, not just a dead version. This heap visit is called a **heap fetch** and it causes random I/O.

With the VM, if the all-visible bit is set for a page, PostgreSQL knows without any doubt that every tuple on it is visible to all transactions. It can **skip the heap fetch entirely**. The index scan becomes a true **index-only scan**, reading only the index and never touching the heap.

Key point

On tables with a high all-visible ratio, skipping heap fetches can reduce I/O by 90% or more for the right query patterns. This is one of the most significant performance features in modern PostgreSQL, and it is entirely dependent on vacuum having run recently enough to set the bits.

Autovacuum Skips All-Visible Pages

When autovacuum runs, it checks the VM before reading each heap page. If the all-visible bit is set, that page is skipped entirely.

On a large, mostly-static table where only a small percentage of pages are modified regularly, this means autovacuum processes only the pages that actually need work. Without the VM, autovacuum would have to scan every page every time — prohibitively expensive on large tables.

The All-Frozen Bit Prevents Wraparound Work

Wraparound-prevention vacuum (`VACUUM FREEZE`) must visit heap pages and freeze old tuples. If the all-frozen bit is already set, these pages are skipped.

On large static archive tables, once all pages are frozen, wraparound vacuum has zero work to do. On active OLTP tables, `autovacuum_freeze_max_age` controls how aggressively PostgreSQL hunts down unfrozen pages.

Why freezing exists at all — and why Bit 2 is not optional decoration — is the circular nature of transaction IDs. The next section is that story.

Why Freezing Exists: Transaction ID Wraparound

Every tuple carries two hidden fields that make MVCC work:

Field	Meaning
xmin	Transaction ID (XID) of the transaction that inserted this row
xmax	XID of the transaction that deleted this row (0 = still alive)

When a transaction reads a row it asks: was the inserting transaction committed before my snapshot? Was the deleting transaction? Those answers come from comparing XIDs. The comparison runs on essentially every row read — which makes the XID counter critically important.

PostgreSQL uses a **32-bit** integer for transaction IDs. The counter runs from 3 up to about 2.1 billion, then wraps back to 3. The design detail that matters: **XID comparison is not linear arithmetic. It is modular (circular) arithmetic.**

PostgreSQL treats the XID space as a circle. Transaction B is “newer than” A if B sits within the forward half of the circle from A — within 2^{31} steps ahead. The same numeric XID can therefore look like the past or the future depending on where the current counter sits.

XID space as a circle (~2.1 billion positions):

~500 million

|

~1 billion ← (current) → 0 / ~2.1 billion

|

~1.5 billion

If current XID = 500,000,000:

XID 200,000,000 → PAST (back half) ✓ visible

XID 800,000,000 → FUTURE (forward half) ✗ not yet visible

XID 1,800,000,000 → PAST (back half) ✓ visible

After wraparound, current XID = 300:

XID 1,800,000,000 → now appears FUTURE ✗ row silently disappears

Without intervention, a row written at XID 1,800,000,000 becomes **invisible after wraparound** — data disappears without an error. This is not theoretical. It has happened on production systems.

Freezing Takes Rows Off the Circle

Freezing is vacuum's answer. When a row's XID is old enough (controlled by `autovacuum_freeze_max_age`, default 200 million transactions), vacuum rewrites `xmin` with a reserved special value:

```
Before freezing:  xmin = 1,800,000,000    xmax = 0
After freezing:   xmin = 2                 xmax = 0
```

The lowest XIDs are reserved:

XID	Meaning
0	Invalid — “no transaction”
1	Bootstrap — used when PostgreSQL initialises itself
2	FrozenTransactionId — permanently visible to everyone
3+	Real transaction counter starts here

`FrozenTransactionId = 2` sits **outside the circular comparison system entirely**. It is never past or future — it is simply always visible. Modular arithmetic cannot reinterpret it.

```
Normal row:  check xmin → circular XID compare → commit log → decide
Frozen row:  check xmin → see "2" → visible. Done.
```

That is a hardcoded rule, not a comparison: if `xmin == 2`, return visible, unconditionally. Because frozen rows no longer hold a real XID, the counter can safely wrap and reuse numbers with no conflict.

Key point

Think of a numbered ticket queue at a bank. Every customer gets a ticket (XID). After two billion customers the machine resets. If old customers

still hold their original numbers, reset causes chaos. Freezing tells old customers to throw the ticket away — they have already been served — so the machine can safely reset.

Once every tuple on a page is frozen, vacuum can set VM Bit 2 (**all-frozen**). Bit 2 is a stronger guarantee than Bit 1: the page needs zero future freeze work, and reads skip both XID comparison and commit-log lookup. It is the fastest possible state for a heap page.

The FSM Enables Efficient Space Reuse

Without the FSM, every `INSERT` would either append to the end of the heap file — causing file growth even when earlier pages have freed space — or scan every page looking for room, which is catastrophically slow.

The FSM answers immediately: *page 4,217 has 3,200 bytes free, use that*. This is why, after a heavy `DELETE` followed by `VACUUM`, subsequent inserts reuse the freed space rather than growing the file.

Caution

The FSM can become stale. If PostgreSQL crashes before vacuum updates it, the map may drive suboptimal space allocation decisions until rebuilt. A regular `VACUUM` always updates the FSM as it goes.

1.3.3 Monitoring

```
-- Vacuum status and dead tuple counts
SELECT relname,
       n_dead_tup,
       last_autovacuum,
       last_autoanalyze
FROM pg_stat_user_tables
ORDER BY n_dead_tup DESC
LIMIT 20;

-- Index-only scan effectiveness.
-- A large gap between idx_tup_read and idx_tup_fetch means many tuples
-- were satisfied without heap fetches: the VM is doing its job.
SELECT relname, idx_scan, idx_tup_read, idx_tup_fetch
FROM pg_stat_user_tables
WHERE relname = 'your_table';

-- Buffer-level I/O split
SELECT relname,
       heap_blks_read, heap_blks_hit,
       idx_blks_read,  idx_blks_hit
FROM pg_statio_user_tables
WHERE relname = 'your_table';

-- Inspect free space per page (requires pg_freespacemap)
CREATE EXTENSION IF NOT EXISTS pg_freespacemap;

SELECT blkno, avail
FROM pg_freespace('your_table')
ORDER BY avail DESC
LIMIT 20;

-- Inspect xmin / xmax (the XIDs that make MVCC and wraparound real)
SELECT xmin, xmax, * FROM your_table LIMIT 10;

-- Force-freeze a static or archive table (sets FrozenTransactionId,
-- enables VM all-frozen bits as pages qualify)
VACUUM FREEZE your_table;
```

1.3.4 Summary

At a glance

Two fork files sit beside every heap.

`<relfilenode>_vm` and `<relfilenode>_fsm` are always present and always maintained. Neither is optional or configurable.

The VM stores two bits per page.

all-visible means every tuple is visible to every transaction; **all-frozen** means every tuple has been frozen against transaction ID wraparound.

The all-visible bit is fragile.

Any insert, update, or delete on a page clears it immediately. It only comes back when vacuum next processes the page.

Index-only scans depend entirely on the VM.

Without the all-visible bit set, every index match still requires a heap fetch to check visibility. With it, the heap is never touched — often a 90% I/O reduction.

The VM is what makes autovacuum affordable.

Autovacuum skips all-visible pages, so a mostly-static table costs work proportional to what changed, not to its total size.

The all-frozen bit eliminates wraparound work.

Once a static table is fully frozen, `VACUUM FREEZE` has nothing to do. `autovacuum_freeze_max_age` governs how aggressively unfrozen pages are hunted down.

XID comparison is circular, not linear.

Transaction IDs are 32-bit and wrap. “Newer than” means within the forward half of the circle. Without freezing, an old `xmin` can look like the future after wraparound and the row silently disappears.

`FrozenTransactionId = 2` leaves the circle.

Vacuum rewrites old `xmin` values to 2. That marker is always visible — no modular compare, no commit-log lookup — so XIDs can be reused safely. Then VM Bit 2 can be set.

The FSM is a tree, not a list.

Parent nodes hold the maximum free space of their children, so a page with enough room is found in $O(\log N)$ rather than by scanning.

The FSM is why space gets reused.

Without it, inserts would append forever and the file would grow even with free space available. It can go stale after a crash; a regular `VACUUM` rebuilds it.

Watch the read/fetch gap.

A large gap between `idx_tup_read` and `idx_tup_fetch` in `pg_stat_user_tables` means the VM is working.

1.4 The Buffer Pool and Clock-Sweep Eviction

Every page PostgreSQL touches passes through shared memory. How pages enter, how they are chosen for eviction, and why “25% of RAM” is a starting point rather than an answer.

1.4.1 What It Is

PostgreSQL does not read from or write to disk directly when processing queries. Every heap page and every index page must first be loaded into a region of shared memory called the **shared buffer pool**, configured by `shared_buffers`.

All backends — every connection, every worker process — share this single pool. When PostgreSQL needs a page, it first checks whether it is already resident (a **buffer hit**). If not, it reads the page from disk into a pool slot (a **buffer miss**, requiring I/O).

The pool is made up of **buffer slots**, each exactly one page (8 KB). The number of slots is `shared_buffers` divided by 8 KB. At `shared_buffers = 4GB`, that is 524,288 slots.

Buffer Descriptors

Each buffer slot has an associated **buffer descriptor** in shared memory.

Field	Meaning
Page identity	<code>relfilenode</code> , fork, and block number
Pin count	How many backends are actively using this buffer. A pinned buffer is never evictable.
Usage count	0 to 5 — how “hot” this buffer is
Dirty flag	Whether the in-memory page has been modified and must eventually be written to disk

Clock-Sweep Eviction

When PostgreSQL needs to load a new page but the pool is full, it must evict an existing buffer. The eviction algorithm is **clock-sweep**, a hardware-efficient approximation of least-recently-used.

Imagine all buffer descriptors arranged in a circular ring, with a single **clock hand** sweeping continuously around it.

1. The clock hand advances to the next buffer descriptor.
2. If the buffer is **pinned** — actively in use — skip it. Pinned buffers are never evicted.
3. If the usage count is **greater than 0**, decrement it by one and advance. This buffer gets another chance.
4. If the usage count is **0**, this buffer is a candidate for eviction. If it is dirty, write it to disk first (or hand it to the background writer), then evict it and claim the slot.

Usage counts are incremented, up to a maximum of 5, each time a buffer is accessed. A frequently-accessed buffer accumulates a high usage count and survives many sweeps. A buffer accessed once and never again decays to 0 quickly and is evicted at the first opportunity.

Key point

The implementation lives in `src/backend/storage/buffer/freelist.c` and is deliberately simple. Evaluating buffer descriptors potentially millions of times per second requires something fast and lock-friendly — not a perfect LRU, which would need a globally sorted list under a lock on every single access. That would be a concurrency disaster.

1.4.2 Why This Matters

`shared_buffers` Is the Highest-Leverage Parameter

The conventional wisdom of “set it to 25% of RAM” is a reasonable starting point, but the mechanics behind it matter far more than the number.

PostgreSQL does not bypass the operating system’s own page cache. When it reads a page from disk into a buffer slot, the OS also caches that read in its kernel page cache. If PostgreSQL later evicts the buffer and then needs the page again, it issues a `read()` call — and the OS often serves it from the kernel page cache rather than physical disk.

Effectively, PostgreSQL benefits from **two levels of caching**: its own buffer pool and the OS page cache beneath it. Setting `shared_buffers` too high steals memory from the second level.

Workload	Implication
Large sequential scans, reporting, bulk loads	OS readahead and page cache are often more effective than the buffer pool. Setting <code>shared_buffers</code> to 80% of RAM frequently performs <i>worse</i> than 25%, because the OS has no memory left for the sequential reads that dominate.
OLTP with a hot random-access workload	A larger <code>shared_buffers</code> helps. The same pages are accessed repeatedly and clock-sweep correctly identifies and retains them.

Scan Pollution

Large sequential scans can **pollute the buffer pool**, evicting hot index pages that should stay cached. Clock-sweep's scan resistance is imperfect — a full table scan rapidly increments usage counts for every page it touches, potentially displacing frequently-used OLTP pages.

`pg_prewarm` and careful use of `enable_seqscan` strategies exist partly to manage this.

The Background Writer Keeps the Clock Hand Moving

The **background writer** (`bgwriter`) runs continuously, writing **dirty buffers** to disk in small batches. Its purpose is to prevent the clock-sweep from having to perform expensive synchronous writes when it encounters a dirty buffer during eviction.

Caution

If `bgwriter_lru_maxpages` and `bgwriter_lru_multiplier` are tuned too conservatively, the clock-sweep ends up writing dirty pages itself — and that latency lands on whatever query happened to trigger the eviction. Watch `pg_stat_bgwriter.buffer_backend`: if it is high, `bgwriter` is not keeping up.

1.4.3 Monitoring

```

CREATE EXTENSION IF NOT EXISTS pg_buffercache;

-- What is actually in the buffer pool right now?
SELECT c.relname,
       count(*) AS buffers,
       round(100.0 * count(*)
            / (SELECT count(*) FROM pg_buffercache), 2) AS pct_of_pool
FROM pg_buffercache b
JOIN pg_class c ON c.relfilenode = b.relfilenode
GROUP BY c.relname
ORDER BY buffers DESC
LIMIT 20;

-- Usage-count distribution: how hot is the pool?
SELECT usagename, count(*) AS buffers
FROM pg_buffercache
WHERE relfilenode IS NOT NULL
GROUP BY usagename
ORDER BY usagename;

-- Background writer activity
SELECT buffers_clean,
       buffers_backend,
       buffers_checkpoint,
       maxwritten_clean
FROM pg_stat_bgwriter;

```

A table consuming 40% of the buffer pool with a usage count near 1 — accessed once, then cold — is polluting the pool. Large sequential scans are the most common cause.

1.4.4 Summary

At a glance

Nothing reaches disk directly.

Every heap and index page passes through the shared buffer pool. A page already resident is a buffer hit; anything else costs I/O.

The pool is a fixed array of 8 KB slots.

Slot count is `shared_buffers` divided by 8 KB, and every backend shares the same pool.

Each slot carries a descriptor.

Page identity, pin count, usage count (0–5), and a dirty flag. Pinned buffers can never be evicted.

Clock-sweep approximates LRU cheaply.

A hand circles the descriptors, decrementing usage counts and evicting the first unpinned buffer that reaches zero. A true LRU would need a global lock on every access.

There are two caches, not one.

The OS page cache sits beneath the buffer pool. Setting `shared_buffers` too high starves the second level, which is why 80% of RAM often performs worse than 25%.

The right size depends on the workload.

Sequential and reporting workloads lean on OS readahead and prefer a smaller pool; OLTP with a hot working set benefits from a larger one.

Sequential scans pollute the pool.

A full table scan raises usage counts on pages it will never revisit, displacing hot index pages. Scan resistance is imperfect.

The background writer absorbs write latency.

If `pg_stat_bgwriter.buffers_backend` is high, bgwriter is not keeping up and eviction cost is landing on user queries.

Inspect the pool with `pg_buffercache`.

A large relation sitting at usage count 1 is the signature of pollution.

1.5 WAL Internals

How PostgreSQL stays durable without paying a random write for every change — and why the same sequential log that survives a crash is the stream that powers replicas, logical decoding, and the disk-full emergency you will eventually debug at 2 a.m.

1.5.1 What It Is

Every database faces the same uncomfortable truth: the CPU and RAM operate in nanoseconds, but the disk operates in milliseconds. That gap is six orders of magnitude. If every change had to be written to its final location on disk the moment it happened — and `fsync'd` to guarantee it reached durable storage, not just the OS cache — a single `INSERT` would cost a seek, a random write, and a forced flush. Do that for every row in every transaction and the system can barely keep up with a moderate workload.

The industry's answer is the **write-ahead log**. Instead of writing each change to its final page immediately, PostgreSQL first **appends** a compact record of the change to a log. Appending is always sequential — the log grows at the end, no seeking around. The actual pages stay in the shared buffer pool and flush to disk later, in larger, cheaper batches.

The “write-ahead” part is the durability contract: **the log record must reach disk before the client is told the transaction succeeded**. If the system crashes a millisecond later, nothing is lost. On startup PostgreSQL reads the log, finds what was recorded but not yet reflected in the data files, and replays those records. The RAM did not survive. The log did.

Key point

Sequential writes are dramatically cheaper than random writes. That single fact is why WAL exists, why Kafka looks the way it does, and why a crash can be recovered from a tape of appends rather than a scatter of page writes.

WAL Is Everywhere

The pattern is universal because the problem is universal.

System	What they call it
PostgreSQL	WAL
MySQL / InnoDB	Redo log
SQLite	WAL-mode journal
MongoDB (WiredTiger)	Journal
Kafka	The product <i>is</i> a WAL — producers append, consumers read
RocksDB / Cassandra / LevelDB	WAL plus memtable (LSM-tree)

Kafka does not hide the log behind an abstraction — the log *is* the API. Producers append; consumers track an offset and read forward. LSM-tree engines use the same RAM-buffer-with-logged-durability pattern: every write goes to a WAL and into an in-memory memtable; when the memtable fills it flushes as a sorted file. The WAL exists only to survive the crash before that flush.

The idea predates computers. It is how accountants kept ledgers: write what you are about to do *before* you do it, so you can always recover.

What Is an LSN?

Every byte written to the WAL has an address. That address is an **LSN** — a **log sequence number**. It is a 64-bit integer: an offset into the WAL stream, measured in bytes from the beginning of WAL for this cluster.

Think of the entire WAL stream as one infinitely long tape. The LSN is the position on that tape.

LSN: 0/3A1F2B40

| byte offset within the WAL stream
 └─── high-order segment counter

LSN 0/3A1F2B40 means “byte 974,796,608 from the start.” As more WAL is written the current LSN advances and **never goes backward**.

LSNs appear everywhere because they put everything on the same timeline:

- A **checkpoint** records its LSN: all data changes before this point are safely on disk.
- A **replica** reports its LSN: it has received and applied WAL up to this point.
- **Replication lag** is the LSN difference between primary and replica, converted to bytes.
- Each **data page** stores the LSN of the last WAL record that modified it. Recovery uses this to skip records already reflected on disk.
- **Point-in-time recovery** says “restore to LSN *X*” — replay until that position and stop.

```
-- Current WAL position on the primary
SELECT pg_current_wal_lsn();
-- → 0/3A1F2B40

-- Which WAL file contains that LSN
SELECT pg_walfile_name('0/3A1F2B40');
-- → 0000000100000000000000003

-- Replication lag in bytes
SELECT pg_wal_lsn_diff(pg_current_wal_lsn(), replay_lsn) AS lag_bytes
FROM pg_stat_replication;
```

WAL Segments

WAL is a continuous stream of records, stored physically as fixed-size files called **WAL segments** — **16 MB** each by default (configurable at `initdb` with `--wal-segsize`). They live in `$PGDATA/pg_wal/` and the filename encodes their position:

```
00000001000000000000000001
00000001000000000000000002
00000001000000000000000003
|           |           | segment number (hex)
|           |           | log file number (hex)
|           |           | timeline ID
```

The **timeline ID** matters after a failover or point-in-time recovery. It prevents a recovered replica from accidentally replaying WAL from a different branch of history.

Old segments are **recycled** — renamed for reuse — after a checkpoint confirms that every change they describe has been flushed to the data files. They are held longer if:

- **Archiving** is enabled — the segment is not recycled until `archive_command` confirms it was archived.
- A **replication slot** is active — the segment is held until the slot's consumer confirms it received and applied the records.

wal_level

`wal_level` controls how much information goes into each WAL record. More information means more WAL volume, and more features.

Level	Use case
<code>minimal</code>	Crash recovery only. Some bulk operations bypass WAL entirely. Does not support replication.
<code>replica</code>	Streaming replication. Default since PostgreSQL 10. Enables <code>pg_basebackup</code> .
<code>logical</code>	Logical replication and decoding. Required for publications, Debezium, pglogical, and AWS DMS.

The volume difference between `replica` and `logical` is usually small. The extra data `logical` writes is old row values — the before-image that logical decoding needs to reconstruct an `UPDATE`.

Checkpoint Anatomy

WAL is the safety net, but replaying it has a cost. If PostgreSQL had to replay *all* WAL from the beginning of time after every crash, recovery could take hours on a busy system.

A **checkpoint** is a known-good point on the WAL timeline: everything before this LSN is already safely reflected in the data files. If you crash, only replay from here forward.

A checkpoint is a promise: **all dirty pages that existed in the shared buffer pool at the moment this checkpoint started have been written to the data files and fsynced to durable storage.**

A dedicated background process, the **checkpointer**, runs this automatically. Its job is the shared buffer pool — dirty pages in RAM that have not yet been written to the data files. WAL is involved twice: at the start, to record the **redo point LSN** (“if we crash mid-checkpoint, where does recovery start?”), and at the end, to announce completion and update `pg_control` with the new safe LSN. Everything in between is buffer-pool work.

The flush is deliberately spread across most of the checkpoint interval — `checkpoint_completion_target`, default 0.9 — rather than done all at once.

Flushing thousands of dirty pages simultaneously would spike I/O into query latency. Once every dirty page is written, the checkpointer calls `fsync()` to force those writes past the OS cache. That is the step that makes the durability guarantee real. After `fsync`, old WAL segments before the redo point are recycled or archived.

Key point

A checkpoint works on the buffer pool. WAL is the before-and-after announcement, and the tool recovery uses afterwards if things go wrong.

1.5.2 Why This Matters

Once you have a sequential, ordered, complete record of every change, you can use it for more than crash recovery.

Streaming and Logical Replication

A replica can connect to the primary, receive the same WAL stream, and apply the same records in the same order. Because the WAL describes changes at the **physical** level — which bytes changed on which page — the replica ends up **byte-for-byte identical**. That is what powers read replicas and standby servers.

Physical records say things like: “bytes 42–50 of page 7 of relation 16384 changed to these values.” That is useless to Kafka or a downstream database. **Logical decoding** reads the same stream and reconstructs it at a higher level: “row with `id=5` in table `orders` was updated, `status` changed from `pending` to `paid`.”

That is what `wal_level = logical` enables, and what Debezium, pglogical, and AWS DMS consume.

Every Write Touches WAL

INSERT, UPDATE, DELETE, DDL, even some VACUUM operations — they all generate WAL records. WAL write performance is a direct limit on write throughput.

`synchronous_commit` is the durability-versus-performance knob.

Setting	Behaviour
on	Default. WAL is flushed to disk before the client gets a success response. Full durability.
off	The server acknowledges the commit before WAL is flushed. Risk window: up to <code>wal_writer_delay</code> × 3, typically a few hundred milliseconds. Significant throughput gains.

`synchronous_commit = off` is appropriate for workloads like metrics ingestion, where losing a few hundred milliseconds of recent commits on a crash is acceptable.

The Checkpoint Frequency Trade-off

Dirty pages accumulate in RAM continuously. The question is not whether to flush them — it is how often, and how aggressively. Both extremes hurt: too frequent, and background I/O constantly competes with queries; too aggressive per checkpoint, and a spike stalls everything else on the disk.

The goal is infrequent checkpoints whose I/O is spread smoothly across the full interval.

More frequent	Less frequent
Less WAL to replay after a crash	More WAL to replay after a crash
More background write I/O	Less background write I/O
More full-page writes	Fewer full-page writes

Two parameters control when a checkpoint fires — whichever threshold is hit first:

- `checkpoint_timeout` (default 5min) — time since the last checkpoint.

- **max_wal_size** (default 1GB) — WAL generated since the last checkpoint.

The defaults are conservative. On a write-heavy production system the typical failure mode is checkpoints firing too often and doing too much I/O in a short burst. Four knobs address this together:

```
# postgresql.conf

# Batch WAL writes in memory before flushing to disk.
# The default (~4MB) is too small for write-heavy workloads.
wal_buffers = 64MB

# Spread checkpoint I/O across 90% of the interval
# instead of a burst at the start.
checkpoint_completion_target = 0.9

# More time between checkpoints: fewer I/O bursts.
# Trade-off: longer recovery if you crash.
checkpoint_timeout = 15min

# Allow WAL to grow larger before forcing a checkpoint.
max_wal_size = 4GB
```

Each knob does a different job. **wal_buffers** batches WAL writes so flushes are fewer and larger. **checkpoint_completion_target** spreads the I/O of a single checkpoint across the interval. **checkpoint_timeout** and **max_wal_size** decide how often a checkpoint starts. Tuning one without the others gives partial benefit at best.

pg_wal Filling Up Is an Emergency

WAL segments accumulate when they cannot be recycled.

Trap

A **replication slot with no active consumer** holds WAL back indefinitely, with no size limit by default. **pg_wal** grows until the disk is full. Set **max_slot_wal_keep_size** as a hard cap on every slot.

The other two common causes: **archive_command** failing, so PostgreSQL refuses to recycle unarchived segments; and a very long-running transaction, which indirectly stalls replication-slot advancement.

1.5.3 Monitoring

```
-- Current WAL position and file name
SELECT pg_current_wal_lsn(), pg_walfile_name(pg_current_wal_lsn());

-- Checkpoint activity and I/O stats
SELECT checkpoints_timed,
       checkpoints_req,
       buffers_checkpoint,
       checkpoint_write_time,
       checkpoint_sync_time
FROM pg_stat_bgwriter;

-- Total WAL on disk
SELECT count(*), pg_size_pretty(sum(size))
FROM pg_ls_waldir();

-- Detect dangerous inactive replication slots
SELECT slot_name, active, restart_lsn, confirmed_flush_lsn
FROM pg_replication_slots
WHERE active = false;
```

On PostgreSQL 17 the checkpoint counters moved from `pg_stat_bgwriter` to `pg_stat_checkpointer`. The columns are the same idea; the view is not.

1.5.4 Summary

At a glance

The log is written first.

Every change is appended to WAL before the data page is flushed. Sequential appends are cheap; random page writes are not. That is the whole reason WAL exists.

Durability is a contract about the log, not the heap.

The client is told “committed” only after the WAL record is on disk. Crash recovery replays the log; the buffer pool does not have to have survived.

An LSN is a byte offset on an infinite tape.

It never goes backward. A checkpoint, a replica, a heap page, and a PITR target are all positions on the same timeline.

Segments are 16 MB files.

They live in `pg_wal` and are recycled after a checkpoint unless archiving or a replication slot still needs them. The filename encodes timeline, log, and segment.

`wal_level` trades volume for features.

`minimal` is crash recovery only. `replica` enables streaming. `logical` adds before-images so decoding can reconstruct row changes.

A checkpoint is a known-good LSN.

The checkpointer's real work is flushing dirty buffers. WAL records the redo point at the start and the completion at the end.

Tune the four knobs together.

Buffer size, completion target, timeout, and max WAL size. One without the others is a partial fix.

`synchronous_commit` is the durability knob.

`off` buys throughput by acknowledging before the WAL flush. The risk window is a few hundred milliseconds.

An idle replication slot fills the disk.

There is no default size limit. `max_slot_wal_keep_size` is the safety catch. Failed `archive_command` is the other classic cause.

1.6 Full-Page Writes

Why WAL sometimes carries an entire 8 KB page instead of a tiny delta — and why turning that behaviour off on the wrong storage is a data corruption risk waiting for a power failure.

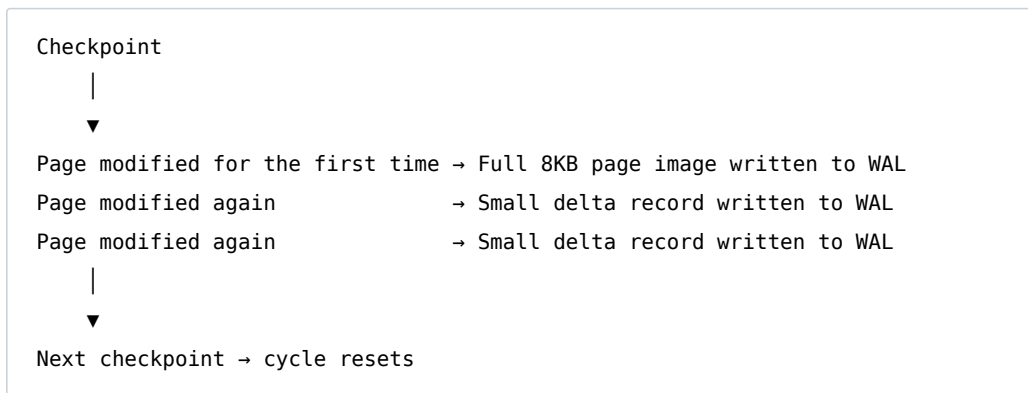
1.6.1 What It Is

WAL records are usually small. A typical delta says something like: “on page 4,217 at offset 312, change these 20 bytes from X to Y.” That is enough for crash recovery when the page on disk is intact: read the page, apply the patch, move on.

But what if the page on disk is *not* intact?

After every checkpoint, the **first time** PostgreSQL modifies a heap or index page it writes the **entire** 8 KB page into the WAL stream. Every later change to that same page *during the same checkpoint interval* writes only a delta. This behaviour is called **full-page writes** (FPW), controlled by `full_page_writes` (default: on).

The full-page image is not written on every modification. It is written once per page per checkpoint cycle. After that initial image is in WAL, subsequent changes to the same page are cheap delta records until the next checkpoint resets the cycle.



The Problem: A Checkpoint Can Leave a Torn Page on Disk

During a checkpoint the checkpointer flushes dirty buffers from the shared buffer pool to the data files on disk. That flush is not one atomic operation for the whole 8 KB page.

Storage devices historically used 512-byte sectors (modern “Advanced Format” drives use 4096-byte sectors — still smaller than PostgreSQL’s 8 KB page). A single logical `write()` for 8 KB may complete only some sectors before power loss. The OS and filesystem do not guarantee atomic 8 KB writes.

If the system crashes mid-flush, the page on disk can end up half old and half new — neither version coherent. That is a **torn page**.

Key point

The torn page problem is not theoretical. It happens when a checkpoint (or any large page write) is interrupted partway through. The disk holds a page that never existed in memory as a whole.

Why a WAL Delta Alone Cannot Fix It

Crash recovery replays WAL by reading a page from disk and applying delta records on top of it. That works only when the base page is trustworthy.

A delta record assumes a valid starting point: “change these 20 bytes at offset 312.” If the page on disk is torn — a mix of old and new sectors — there is no valid base to patch. Applying the delta on top of corrupt data produces garbage. The database is corrupted in a way no amount of replay can salvage from deltas alone.

This is the gap FPW closes. Deltas are efficient for normal operation, but they cannot recover from a page that was never written atomically to disk.

The FPW Solution: A Clean Copy in WAL Becomes the Base

Full-page writes solve the torn page problem by putting a complete, self-consistent 8KB page image into WAL immediately after each checkpoint — before any delta records for that page in the new interval.

During crash recovery PostgreSQL does not trust the on-disk version of a page that has a full-page image in WAL. It **uses the image from WAL as the base** instead of reading the possibly torn page from disk. Any delta records that follow are applied on top of that clean copy.

The page ends up correct regardless of what was on disk when the crash happened. The full-page image is the insurance policy against a checkpoint interrupted mid-write.

1.6.2 Why This Matters

WAL Write Amplification

Full-page writes are a significant source of **WAL write amplification**. Immediately after a checkpoint, every page that is modified generates an 8 KB WAL record instead of a tiny delta. On a write-heavy OLTP system with frequent checkpoints, this dramatically increases WAL generation rate and WAL-related I/O.

To quantify: suppose your workload modifies 50,000 distinct pages right after a checkpoint. That is $50,000 \times 8 \text{ KB} = \mathbf{400 \text{ MB}}$ of full-page images flooding into WAL. As the checkpoint interval progresses and each page gets its initial FPW done, WAL volume drops to delta-only records. You can observe this characteristic sawtooth in WAL generation metrics — volume spikes at each checkpoint and declines until the next one.

`wal_compression` Is Almost Always the Right Lever

Available since PostgreSQL 9.5, `wal_compression` compresses full-page images before writing them to WAL. Full-page images are typically very compressible — much of an 8 KB page is null bytes, padding, or repetitive tuple data. Compression ratios of 5:1 to 10:1 are common, turning an 8 KB full-page image into roughly 800 bytes in WAL.

```
# postgresql.conf

# LZ4 is faster than pglz; available on PostgreSQL 15+
wal_compression = lz4
```

The CPU cost is modest on modern hardware. The reduction in WAL write volume, replication network bandwidth, and WAL archiving storage is substantial. Enable it unless you have a specific reason not to.

What to Actually Do About FPW Overhead

Lever	Effect
<code>wal_compression = lz4</code>	Compresses FPW images; typically 5–10× size reduction with low CPU cost
Raise <code>checkpoint_timeout</code> (e.g. 5→10 min)	Fewer checkpoint cycles → fewer FPW resets
<code>checkpoint_completion_target = 0.9</code>	Smooths checkpoint I/O; does not reduce FPW volume but prevents I/O spikes

These three settings together handle FPW overhead gracefully on virtually all production workloads without sacrificing durability.

Disabling FPW Is Dangerous

Some sources suggest disabling FPW as a performance optimization. This is safe only on storage that guarantees **atomic 8 KB writes** — meaning if a write is partially interrupted, the storage either completes the full write or reverts to the original state. Some enterprise SAN configurations with battery-backed write caches provide this guarantee. Copy-on-write filesystems like ZFS also mitigate the torn page risk.

On commodity hardware with standard Linux `ext4` or `xfs` filesystems, `full_page_writes = off` is **a data corruption risk waiting for a power failure**. The performance benefit does not justify the risk in nearly all real production environments.

1.6.3 Monitoring

```
-- Confirm FPW is enabled (should be 'on' on almost all systems)
SHOW full_page_writes;

-- Check whether WAL compression is on
SHOW wal_compression;

-- Checkpoint frequency drives how often the FPW cycle resets
SELECT checkpoints_timed,
       checkpoints_req,
       checkpoint_write_time
FROM   pg_stat_bgwriter;
```

On PostgreSQL 17 the checkpoint counters moved from `pg_stat_bgwriter` to `pg_stat_checkpointer`. The columns are the same idea; the view is not.

Watch WAL generation rate alongside checkpoint timing. A spike in bytes written to WAL immediately after each checkpoint is the FPW sawtooth pattern. If the spike is painful, `wal_compression` is the first knob; lengthening the checkpoint interval is the second.

1.6.4 Summary

At a glance

A checkpoint can leave a torn page on disk.

An 8 KB flush is not atomic. Power loss mid-write can produce a page that is half old and half new.

WAL deltas need a trustworthy base.

Recovery reads a page from disk and patches it. You cannot apply a delta on top of corrupt data — the result is garbage.

FPW puts a clean copy in WAL.

After each checkpoint, the first modification to each page writes the full 8 KB image to WAL. Recovery uses that image as the base instead of the possibly torn disk page.

FPW happens once per page per checkpoint interval.

After the initial full-page image, subsequent changes to the same page write only deltas until the next checkpoint resets the cycle.

FPW causes WAL write amplification.

Right after a checkpoint, WAL volume spikes as every touched page gets an 8 KB record. The sawtooth pattern is normal.

wal_compression is the first lever.

LZ4 compression typically shrinks full-page images 5–10× with modest CPU cost. Enable it on virtually all production systems.

Do not disable FPW on commodity storage.

`full_page_writes = off` is safe only when the storage guarantees atomic 8 KB writes. On standard Linux filesystems it is a corruption risk.

FPW ties the six storage topics together.

WAL records the change (with FPW after checkpoints); the buffer pool holds the page; the checkpointer flushes it to disk — and FPW exists because that flush might not finish cleanly.

1.7 Chapter Summary

Six topics, one system. A row is written on a heap page, possibly toasted, tracked by the visibility and free-space maps, held in the buffer pool, logged in WAL, and eventually flushed at a checkpoint — with full-page writes guarding against torn pages.

```
Write → WAL (+ full-page image if first since checkpoint)
    → buffer pool (clock-sweep eviction; bgwriter pre-writes dirty pages)
    → heap tuple on 8KB page (MVCC header, line pointer)
    → TOAST if value too large
    → VM bit cleared, FSM updated
    → autovacuum reclaims dead tuples, sets VM, updates FSM
    → index-only scans skip heap via VM; inserts find space via FSM
    → checkpoint flushes all dirty pages → FPW cycle resets → WAL recycled
```

Most important

1.1: Heap file layout.

The page is the unit of I/O — every table is an 8 KB heap file. Tuple headers carry MVCC (`t_xmin`, `t_xmax`); line pointers are the indirection layer indexes use via `ctid`. Updates never modify in place: an `UPDATE` writes a new tuple and marks the old one dead, which is why bloat happens and why indexes store physical addresses.

1.2: TOAST.

A row must fit in one page. Values above roughly 2 KB are compressed or moved to a side `pg_toast` table. Selecting a toasted column is the hidden I/O cost; `pg_column_size()` can mislead because it reports stored size, not read cost.

1.3: Visibility map and free space map.

Every heap has two fork files: `_vm` stores two bits per page (all-visible, all-frozen); `_fsm` is a tree of free-space counts. Index-only scans and autovacuum skip all-visible pages. Inserts consult the FSM to reuse space instead of always appending. XIDs are 32-bit and compared on a **circle**; freezing sets `xmin = 2` so wraparound cannot make old rows vanish, and VM Bit 2 marks pages that need no more freeze work.

1.4: Buffer pool.

Nothing reaches disk directly from a query — every page passes through the shared buffer pool, a fixed array of 8 KB slots (**shared_buffers**). Clock-sweep approximates LRU for eviction when the pool is full. The background writer runs continuously to pre-write dirty pages so eviction stays cheap. These two are easy to confuse with the checkpointer — see the highlight below.

1.5: WAL internals.

WAL is written first — durability is a contract about the log, not the heap. An **LSN** is a byte offset on the sequential WAL tape. Segments are 16 MB files in `pg_wal`, recycled after a checkpoint unless archiving or a replication slot holds them. Tune **wal_buffers**, **checkpoint_timeout**, **max_wal_size**, and **checkpoint_completion_target** together.

1.6: Full-page writes.

Checkpoints can **tear pages** — an 8 KB flush is not atomic. WAL deltas cannot patch corrupt data, so after each checkpoint the first change to a page writes a full clean 8 KB copy into WAL. Enable **wal_compression**; do not disable **full_page_writes** on commodity storage.

The whole chapter in one sentence.

Every performance problem — bloat, slow vacuums, replication lag, checkpoint storms, index-only scan regressions, WAL spikes — traces back to one of these six mechanisms. Once the model is in place, `EXPLAIN ANALYZE`, `pg_stat_*`, and `postgresql.conf` become far more intuitive.

Three mechanisms, one action

The most common confusion in this chapter: **clock-sweep eviction**, the **background writer** (`bgwriter`), and the **checkpointer** can all write dirty pages from the buffer pool to the data files — the same physical work, but for **different reasons**.

Mechanism	What it is actually for
Clock-sweep eviction	Free a buffer slot when the pool is full. Writes a dirty page only if the chosen victim is dirty and has not been flushed yet.
Background writer	Pre-write dirty pages in the background so clock-sweep rarely has to. Hidden from queries when it keeps up.
Checkpoint	Flush <i>every</i> dirty page, <code>fsync</code> , record a known-good WAL point, recycle old WAL. Not eviction — a durability and recovery operation.

Key point

Worst case for clock-sweep: the background writer has fallen behind, the pool is full, and a user query triggers eviction of a dirty page. That query does the synchronous write itself — that is what `pg_stat_bgwriter.buffer_backend` measures. When it climbs, random query latency climbs with it. The fix is not “tune checkpoint”; it is give the background writer more room (`bgwriter_lru_maxpages`, `bgwriter_lru_multiplier`) or reduce pressure on the pool (`shared_buffers`, scan pollution).

Indexing Deep Dive

An index maps a key to ctid. This chapter covers the default sorted map (B-tree), when a read can stop at the leaf, hash buckets for equality, BRIN summaries for append-ordered ranges, GIN inverted lists for many keys per row, GiST trees of predicates for shapes that do not live on one sorted line, a decision framework for choosing among them, then three cuts on the same tree: which rows participate (partial), how many columns share the key (composite), and what value is stored as the key (expression).

2.1 B-Tree and B+ Tree

The heap is unordered. An index is a sorted shortcut into it — and the shape of that shortcut determines everything: how fast a range scan runs, what happens to the structure on every write, and why the key type you choose for a primary key is not a cosmetic decision.

2.1.1 What It Is

Heap Is Unordered by Default

PostgreSQL heap tables are **non-clustered by default**. Rows are written wherever the next free space happens to be — not in any particular key order. A row inserted today might sit on page 1; the next insert could land on page 400. There is no physical ordering by `id`, `created_at`, or anything else.

PostgreSQL does have a `CLUSTER` command that physically rewrites the heap in index order, but it is a one-time operation. The moment new rows are written after the rewrite, physical order drifts again. Nothing maintains it automatically. For practical purposes, treat the heap as always unordered.

This matters because a query like `WHERE id = 42` cannot skip pages — it has to check every page in the file to be sure it did not miss a match. That is a **sequential scan**, and it is fine for small tables or queries that genuinely need most of the rows. For everything else it is wasteful.

An index is a *separate structure*, maintained alongside the heap, that provides the ordered access the heap itself cannot give. It maps a search key to the physical address (**ctid**) of the matching tuple so the engine can jump straight to it.

Key point

The heap does not sort your rows. The index sorts the keys in a separate structure and points back to the heap. That separation is why updates that do not change indexed columns can sometimes skip updating the index at all — the heap moved, but the sorted map did not need to change.

Because disk is slow, that sorted map must be **wide, shallow, and balanced**:

- **Wide** — each node holds many keys. One page read loads an entire node, so more keys per node means fewer reads to reach any key.
- **Shallow** — with many keys per node the tree needs very few levels. A million-row index is typically three or four levels deep.
- **Balanced** — every leaf sits at the same depth. No search path is longer than any other.

That is the B-Tree family.

Inside an Index Page

Chapter 1 showed that every heap page shares the same layout: a 24-byte header, a line pointer array, free space in the middle, and tuple data growing up from the bottom. Index pages use the *same* 8 KB format. Two things differ.

Page Header (same as heap)
Line Pointer Array (same as heap)
Free Space
Index Tuples (key + ctid, no xmin/xmax)
Special Area
– prev page (left sibling)
– next page (right sibling)
– level (0 = leaf, 1+ = internal)

The special area is filled in. Heap pages leave it empty. B-Tree pages use it for three fields: left sibling, right sibling, and level. The sibling pointers *are* the linked leaf chain — a range scan follows next-page one step at a time with no need to climb back up. The level field tells the engine instantly whether a page is a leaf or an internal routing node.

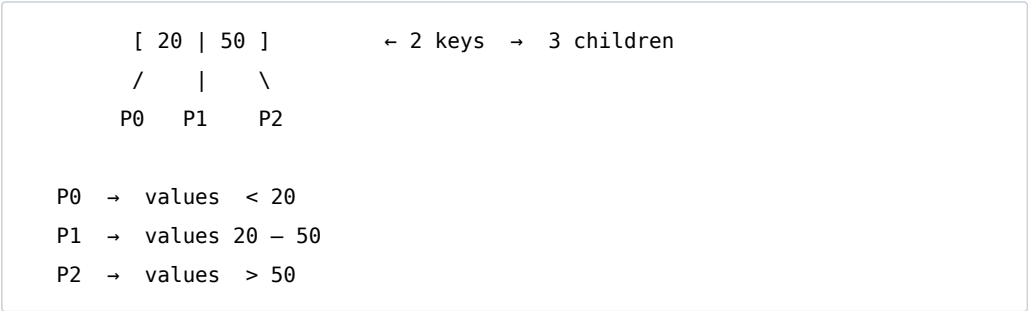
Index tuples carry no visibility information. A heap tuple has `xmin` and `xmax`. Index tuples store only the **search key** and a `ctid`. There is no `xmin`, no `xmax`. Visibility is always resolved by following the `ctid` back to the heap.

Key point

Index pages are the same 8 KB pages as heap pages. The special area now holds sibling pointers (the leaf chain) and a level indicator. Index tuples hold only key and `ctid` — all visibility work stays on the heap.

Node Shape

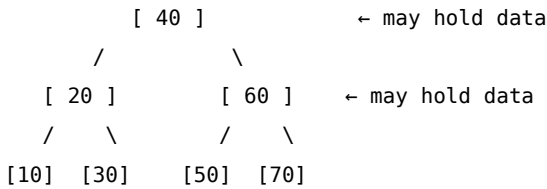
Keys are sorted **fences**. A node with n keys has $n + 1$ children — one slot per side of every fence:



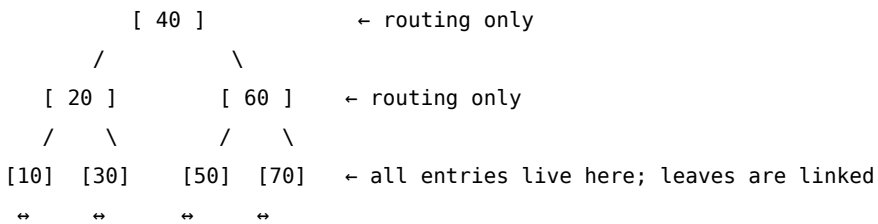
Layer	Role
Root	Single entry point for every search
Internal	Routing only; keys and child pointers
Leaf	Bottom layer; all leaves at the same depth

B-Tree vs B+ Tree

The classic **B-Tree** stores data at *every* level. Leaves are not linked.



A **B+ Tree** keeps *only routing keys* in internal nodes. All data lives in the leaves, and leaves are linked in key order:



	B-Tree	B+ Tree
Data location	Every node	Leaves only
Point lookup	Can stop early	Always to a leaf
Leaves linked	No	Yes
Range scan	Climb around tree	Walk the leaf chain

Key point

PostgreSQL's `CREATE INDEX` produces what the documentation calls a B-Tree. Under the hood it is a B+ Tree: row pointers live only in the leaves, leaves are linked, and internal nodes carry only routing keys.

2.1.2 Why This Matters

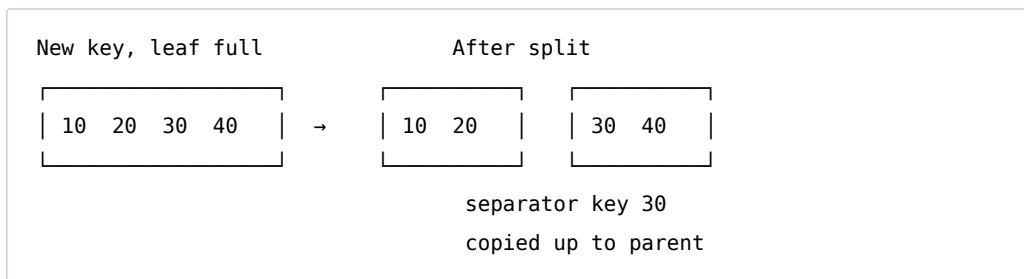
How the Tree Grows

The tree grows by **splitting**. There are two splits. They do not cost the same thing.

When a leaf has free space, either insert is cheap: drop the entry in sorted position and done. Internal nodes are untouched. When the leaf is full, which split you are in depends on the key.

Same key, new value. Another `ctid` for a key the leaf already holds. The separator is already in the parent. If the leaf must split, it cuts that key's `ctids` in two. No new separator key is copied up. Depth does not change.

New key. A distinct value the parent cannot yet separate. This is the split that grows the tree:



The page divides into two halves. The smallest key of the new right leaf — the **separator key** — is copied up into the parent. If the parent is also full it splits in turn, cascading upward until a node with room is found.

The tree gains height only when the *root* splits: a new root is created above it, and every leaf is pushed down one level together. A same-key split never does that.

PostgreSQL tries not to reach either split. A full leaf first deletes known-dead entries, then packs duplicate keys. The next topic covers that order, and the cost of the new-key path under concurrency. The picture above is what remains when neither earlier step frees a slot.

Key point

All leaves stay at the same depth because height grows only when the root splits, which pushes every leaf down at once.

Insert and Update

Operation	What happens in the index
INSERT	Find the leaf. Same key: leaf only. New key that does not fit: separator to the parent
UPDATE on an indexed column	Remove old entry, insert new; either leaf may split
UPDATE on a non-indexed column	Often no new index entry if HOT applies

Indexes and MVCC

Chapter 1 established that an `UPDATE` never overwrites in place: it sets `xmax` on the old tuple and writes a new version. An index entry points at **one physical `ctid`** — **one version**, not a logical row.

Non-HOT update — indexed column changed, or the new tuple landed on a different heap page:

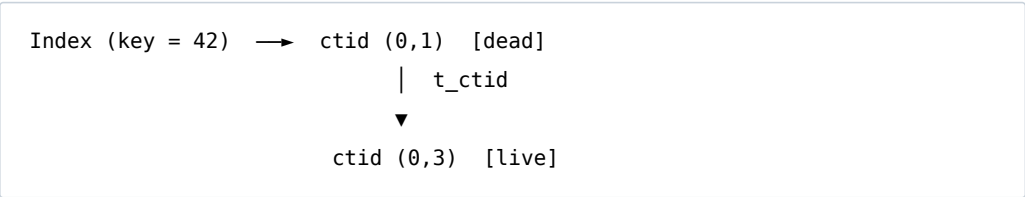
A new index entry is written for the new `ctid`. The old entry stays until vacuum removes it. Temporarily, one key can have several `ctids` in the leaf — some live, some dead.

```
B+Tree leaf (key = 42)
├─ ctid (0,1)  ← dead version
├─ ctid (0,2)  ← dead version
└─ ctid (0,3)  ← live version
```

A lookup visits each `ctid`, checks visibility at the heap, and keeps the live version. This heap round-trip is **index visibility checking**.

HOT update (Heap-Only Tuple) — no indexed column changed *and* the new tuple fits on the same heap page:

No new index entry is written. The index still points at the head of the chain; the heap page walks `t_ctid` to the live version:



	HOT	Non-HOT
New index entries	No	One per version, per index
Index bloat	Lower	Grows with update frequency
Finding live row	Walk HOT chain on the page	Multiple ctids + heap checks

Fillfactor and HOT. The HOT condition requires room on the same heap page. By default PostgreSQL fills heap pages to 100 %. Setting a lower **fillfactor** — say 70 — reserves 30 % of each page for future updates, giving HOT updates somewhere to land instead of spilling to a new page.

```
CREATE TABLE orders (...) WITH (fillfactor = 70);
ALTER TABLE orders SET (fillfactor = 70);
```

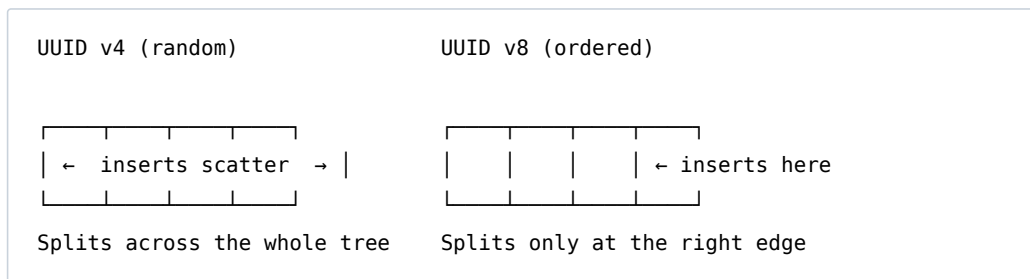
Use it on tables with heavy UPDATE traffic on non-indexed columns. Read-heavy or insert-only tables do not need it.

Key Order and UUID

B+ Tree leaves are sorted. Where a **new key** lands decides how disruptive that split is. A same-key split does not move a separator; these are the inserts that do.

Random keys (UUID v4): each insert targets a random leaf across the whole tree. I/O is scattered, splits occur everywhere, and the buffer cache has no single hot region to warm.

Ordered keys (bigint, IDENTITY, UUID v7/v8): new keys are almost always the largest seen so far. Inserts land at the **rightmost leaf** every time. Splits concentrate at that one edge; the rest of the tree stays cold and undisturbed.



Key point

Same B+ Tree, same algorithm — the key type alone decides whether writes are scattered random I/O or a predictable append at the right edge.

UUID v7 and v8 embed a sortable timestamp in the high-order bits for exactly this reason. On write-heavy tables the choice between v4 and v7/v8 is a real performance decision.

Vacuum and Reindex

Dead leaf entries accumulate as old versions pile up. **Vacuum** removes them surgically, leaving the tree structure intact. This is normal, routine maintenance.

Reindex discards the entire index and rebuilds it from the live heap. It is not a substitute for vacuum — it is a last resort when the tree structure itself has grown bloated or inconsistent. The full treatment comes in a later chapter.

2.1.3 Summary

At a glance

A B+ Tree keeps all data in the leaves.

Internal nodes are routing only. Linked leaves make range scans and ORDER BY a linear walk rather than repeated root-to-leaf descents.

Two splits, not one.

A new value for a key already on the leaf cuts that leaf's `ctids` and leaves the parent alone. A new key copies a separator into the parent. Height grows only when the root splits, so every leaf stays at the same depth.

Index pages are the same 8 KB pages as heap pages.

The special area holds sibling pointers (the leaf chain) and a level indicator. Index tuples hold only key and `ctid` — visibility is always resolved at the heap.

An index entry points at one physical version.

On a non-HOT update a new entry is added for the new `ctid`; old entries linger until vacuum clears them.

HOT skips new index entries.

No indexed column changed and the new tuple fits on the same page: the index does not grow. Lower fillfactor keeps that space available.

Ordered keys concentrate splits at the right edge.

UUID v4 scatters writes across the tree; UUID v7/v8 and sequences append to one edge. Same structure, very different I/O profile.

Vacuum cleans leaves; reindex rebuilds the tree.

Routine maintenance is vacuum. Rebuilding a bloated structure is reindex — used rarely.

2.2 Index-Only Scans and Covering Indexes

The previous topic left the index as a sorted map from key to `ctid`, with no visibility on the leaf. A normal scan still visits the heap. This topic is the exception — and what a full leaf does before it splits.

2.2.1 What It Is

The Ordinary Index Scan

In a normal index scan, every candidate row takes the same path:

```
WHERE user_id = 42

root → internal → leaf
                    |
                    ▼
                ctid (3, 7)
                    |
                    ▼
        heap page 3, line 7
        check xmin / xmax
        return the row
```

The tree walk is a few page reads. The heap fetch is a random read into a separate file, and it happens once per candidate. On a selective query that is still the right plan. On a query that only needed two columns already sitting in the leaf, every heap visit is wasted I/O.

The index answered *where might this row be?* The heap answers *is this version visible, and what are the other columns?*

When the Planner Can Skip the Heap

An **index-only scan** is a plan node, not a property you set on the index. The planner picks it when every column the query touches — WHERE, SELECT, ORDER BY — is available from that index alone.

Two conditions, both required:

Condition	If it fails
The index contains every column the query needs	The planner cannot choose Index Only Scan. It must visit the heap for the missing columns.
The heap page’s all-visible bit is set	The plan node is still called Index Only Scan, but each row does a heap fetch. The name is the plan shape, not a promise.

The visibility map stores one bit per heap page. Vacuum sets it when every tuple on the page is visible to every current and future transaction. Any insert, update, or delete on the page clears it.

Index Only Scan

|

▼

VM bit for this heap page?

set → return the leaf tuple. No heap I/O.

clear → fetch the heap tuple. Heap Fetches += 1

EXPLAIN (ANALYZE) reports this as Heap Fetches. That counter is the whole short-cut in one number.

Index Only Scan using orders_user_covering on orders
Index Cond: (user_id = 42)
Heap Fetches: 1180

Heap Fetches: 0 means the scan did not touch the heap. Heap Fetches close to the row count means the plan looks index-only and behaved like a normal index scan. You paid for the plan shape and still did the random I/O.

Trap

Index Only Scan is a plan shape. Heap Fetches is whether it kept the promise. A high count is not fixed by a wider index. Vacuum sets the all-visible bits; writes and old snapshots clear or block them.

2.2.2 Why This Matters

Why Vacuum Decides

The visibility map is not updated by the query. A table that is written steadily has most recently touched pages with the bit off. Autovacuum turns it back on, and only for pages where no open snapshot could still need an older version.

A long-running transaction holds the xmin horizon back. Vacuum runs, cannot mark those pages all-visible, and the next index-only scan keeps fetching the heap. The index is fine. The snapshot horizon is the problem.

Covering Indexes

A query that selects a column the index does not store cannot be index-only. The tempting fix is to add that column to the key:

```
CREATE INDEX orders_user_status_idx
ON orders (user_id, status, total);
```

That works, and it changes the index. `status` and `total` become sort keys. They occupy internal pages, they participate in comparisons, and on a unique index they become part of what uniqueness means.

`INCLUDE` (PostgreSQL 11+) stores extra columns **only in the leaves**. They are payload. They are not fences.

```
CREATE INDEX orders_user_covering
ON orders (user_id)
INCLUDE (status, total);
```

Internal pages	Leaves
<hr/>	<hr/>
user_id only (routing)	user_id + status, total + ctid (key) (payload) (address)

The tree stays as narrow as a user_id index. A query that filters on user_id and returns status and total can be index-only. A unique index on user_id stays unique on user_id alone — the included columns are not part of the constraint.

Key column versus included column.

You need to...	Where it goes
Search, range, or ORDER BY	Key column, left of INCLUDE
Enforce uniqueness	Key column
Only return it, never search	INCLUDE
Both search and return	Key column. Already in the leaf.

INCLUDE is not a second index. The planner descends using the key columns, then reads the payload that is already there. A predicate on an included column is a filter after the leaf is found, not a descent. If status is how you search, it belongs in the key:

```
-- seek by user_id, return status
CREATE INDEX ON orders (user_id) INCLUDE (status);

-- seek by user_id AND status
CREATE INDEX ON orders (user_id, status);
```

Column order in the key is unchanged: equality columns first, then the range or ORDER BY column. INCLUDE only keeps “columns I merely want back” out of the sort key.

The size limit did not go away. An index tuple still has to fit in about one third of an 8 KB page. Included columns count. A text or jsonb value that is fine in the heap can make CREATE INDEX fail, or force a narrower expression. Covering is for columns you were going to fetch anyway, not for stuffing the whole row into the leaf.

Before a Leaf Splits

The previous topic drew the two splits. A same-key split stays on the leaf. A new key copies a separator into the parent. In PostgreSQL either split is the **last step**, not the first.

When an incoming tuple does not fit, the leaf tries to make room:

1. **Delete entries already known dead** on that page.
2. **Deduplicate**, if the page is still full and the index allows it. Duplicate keys are packed so the key is stored once.
3. **Split**, if neither freed a slot.

That order is why a leaf that looks full of the same key often does not split as soon as the diagram in the previous topic suggests.

Dead entries. Passing scans can mark entries that no snapshot still needs. Since PostgreSQL 14, a page filling up with old versions from updates that did *not* change this index's key can also get a targeted cleanup (**bottom-up deletion**) at insert time — not during `VACUUM`.

A `ctid` on that page is dead when the transaction that created it aborted, or when it committed and the row was later deleted or updated and no live snapshot still needs it. An entry whose transaction is still open stays.

<code>ctid (0,1)</code>	<code>committed, row deleted</code>	<code>dead</code>	<code>-- remove</code>
<code>ctid (0,4)</code>	<code>aborted</code>	<code>dead</code>	<code>-- remove</code>
<code>ctid (0,7)</code>	<code>transaction still open</code>	<code>keep</code>	

PostgreSQL uses that transaction status to decide what is safe to drop. The tree structure does not change: no parent update, no new page. The new key just needs a free slot.

Deduplication. A duplicate is two leaf entries whose key columns match. Logically the leaf holds separate lines. With deduplication (PostgreSQL 13+, on by default) the physical leaf can store one **posting list**:

Logical entries	Posting list
key = paid → ctid (0,1)	key = paid → [(0,1),
key = paid → ctid (0,4)	(0,4),
key = paid → ctid (1,2)	(1,2)]

The key bytes appear once. The `ctids` stay sorted. Visibility is unchanged: each `ctid` is still checked through the heap or the visibility map. Deduplication is a packing trick, not a different index.

Leaves stay sorted, so duplicate keys land next to each other. That is why a single pass can find them and pack them.

```
Before insert
[ failed→(0,1) | paid→(0,4) | pending→(1,2) ]

INSERT paid → (2,1)

After insert (duplicates now adjacent)
[ failed→(0,1) | paid→(0,4) | paid→(2,1) | pending→(1,2) ]
```

The pass is lazy. A new duplicate is not merged on every insert. The page stays in the ordinary representation until an insert would not fit and deleting known-dead entries was not enough. `CREATE INDEX` and `REINDEX` are the exception: they pack posting lists as they build, because the keys are already sorted in one pass.

Reading a posting list is as cheap as reading plain entries. A write-heavy index with almost no duplicates pays a small fixed cost for a pass that finds nothing to merge. Most indexes should leave it on.

```
CREATE INDEX orders_status_idx
ON orders (status)
WITH (deduplicate_items = on); -- the default
```

Situation	Dedup
Low-cardinality key (<i>status</i> , <i>boolean</i>)	The usual win. One key, many <i>ctids</i> , a smaller leaf.
Non-unique index with repeated values	Same benefit.
Unique index with version churn	Packs not-yet-gone versions of the same key so the page does not split while a long transaction pins them. Two live rows still cannot share a unique key.
Unique key that never repeats	Nothing to merge. Splits still happen at the right edge.
Index with <i>INCLUDE</i> columns	Never. Included payload can differ between rows that share a key.
<i>numeric</i> , <i>float</i> , <i>jsonb</i> , <i>arrays</i> , <i>ranges</i>	Disabled. Values that compare equal are not always byte-identical.
<i>text</i> , <i>nondeterministic collation</i>	Disabled. Case or accent differences must be kept. Deterministic collations are fine.

The unique-index row is the one most people miss. Uniqueness is about live rows. MVCC can still leave several index entries for one key. Dedup plus the split-time cleanup is how a unique index absorbs that burst without splitting on every update. It is not a replacement for HOT or for vacuum. A snapshot that never ends will eventually fill the page anyway.

The split. Only if both earlier steps fail does the leaf divide. Which split that is was drawn in the previous topic and finished at the end of this one.

A new value for a key already on the leaf cuts that leaf's *ctids*. The parent already has the separator. A new key copies a separator into the parent, and that is the split that can cascade to the root.

Key point

A full leaf deletes known-dead entries, then packs duplicate keys, and only then splits. `INCLUDE` turns deduplication off. Do not add it just in case on a low-cardinality index you wanted packed.

How the Three Fit

Columns missing from the index

- ordinary index scan, heap fetch every row
- fix: key column, or `INCLUDE` if you only return it

Columns already in the index

- planner may choose Index Only Scan
- Heap Fetches = 0 only while the VM bit is set
- fix a high count with vacuum and shorter transactions

Same key, many ctids, leaf about to split

- delete known-dead entries
- deduplicate into a posting list
- split, last

`INCLUDE` and deduplication pull in opposite directions. A covering index answers more queries from the leaf and gives up posting lists. A deduplicated index is smaller when the key repeats and cannot carry extra payload. Pick the one the workload actually hits: many rows sharing one key, or extra columns in the `SELECT`.

Two Insert Cases

This finishes the split from the previous topic. An insert that still does not fit is one of two cases. The difference is whether the parent already has a separator for that key.

Same key, new value. Another row for a key the leaf already holds — another `ctid` for `paid`, not a key the tree has never seen. The separator is already in the internal node. The insert goes to that leaf.

If the leaf has room, or `dedup` can pack the extra `ctid`, the write stops there. If the leaf is full, it splits. That split stays on the leaf: you are cutting that key's `ctids` in two. You are not installing a new separator key in the parent. Internal nodes stay as they were. Depth does not change.

New key. A distinct value the parent has no separator for. The leaf that should hold it splits, and a new separator key is copied into the parent. That is the expensive path.

If the parent is also full, the split cascades upward. If the cascade reaches the root, the root splits and the tree grows taller. The insert touches the leaf, the new sibling, and every parent on the way up. Those pages are locked. Two sessions doing this on overlapping branches can deadlock. PostgreSQL retries. The retry is still the costly case.

	Same key, new value	New key
Where it goes	The leaf that already holds this key	The leaf for a key the parent cannot yet separate
If the leaf is full	Split that leaf's <code>ctids</code> only	Split the leaf and add a separator to the parent
Internal nodes	Untouched. The separator is already there.	Parent gains a separator. Can cascade.
Depth	Unchanged	Unchanged, until the root itself splits
Concurrency	One leaf	Many pages locked. Deadlock risk on the way up.

The new-key path is the one that hurts under concurrency, and it is worse when the key is not ordered and not expected.

An unordered key (UUID v4) is a new separator in a random place. Splits scatter. Concurrent sessions lock different branches and meet on a shared parent. More pages, more I/O, more deadlocks.

An ordered key (a sequence, UUID v7 or v8) is also a new key, but it is the one the tree expects: larger than anything stored, always at the rightmost leaf. The separator is added, and the next one after it, on that same edge. Sessions contend for one page. They do not carve new separators through the middle of the tree.

Same key, new value

- separator already in the parent

- root → that leaf → insert, or split the leaf's ctids

- internal nodes untouched

New key

- root → leaf → split

- new separator copied into the parent

- parent full? cascade upward

- root splits? tree grows taller

Key point

A new value for a key the leaf already holds splits the leaf, not the tree. A new key adds a separator to the parent, and that is the insert that locks pages on the way up. Unordered keys make that happen everywhere.

2.2.3 Hands-On Inspection

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT status, total
FROM orders
WHERE user_id = 42;
```

Look for Index Only Scan versus Index Scan, and read Heap Fetches. Buffers tell you whether those fetches were cache hits or real reads. A high hit count is still work. It is just cheaper work.

```
SELECT relname,
       relpages,
       relallvisible,
       round(100.0 * relallvisible
            / NULLIF(relpages, 0), 1) AS all_visible_pct
FROM pg_class
WHERE oid = 'orders'::regclass;
```

A low `all_visible_pct` on a table you expected to scan index-only is the usual explanation for a high Heap Fetches count.

```
SELECT c.relname AS index_name,
       i.indnkeyatts AS key_columns,
       i.indnatts AS key_plus_included
FROM pg_index i
JOIN pg_class c ON c.oid = i.indexrelid
WHERE i.indrelid = 'orders'::regclass;
```

`key_plus_included` greater than `key_columns` means the index has `INCLUDE` columns. Those indexes never deduplicate.

```
CREATE INDEX orders_user_covering
ON orders (user_id)
INCLUDE (status, total);
```

After creating it, run the `EXPLAIN` again. If the plan switches to Index Only Scan and Heap Fetches stays high, the index is covering and the visibility map is not. `VACUUM orders`; and a quiet snapshot horizon are what move that counter, not a wider index.

2.2.4 Summary

At a glance

A normal index scan still fetches the heap.

The leaf has a `ctid` and no visibility bits. Each candidate is checked on the heap.

Index Only Scan is a plan node, not an index setting.

The planner can choose it only when every column the query needs is already in the index.

Heap Fetches is the promise.

Zero means the visibility map let the scan skip the heap. A count near the row count means the plan looked index-only and still did the fetches.

Vacuum sets the all-visible bit.

Writes clear it. A long-running transaction can block vacuum from setting it. The fix is vacuum and shorter snapshots, not another index.

INCLUDE stores payload in the leaves only.

Use it for columns you return. Columns you search, range, or `ORDER BY` belong in the key. Included columns do not change uniqueness.

A full leaf does not split first.

It deletes known-dead entries, then deduplicates repeated keys into a posting list, and splits only if it is still full.

Dedup is on by default since PostgreSQL 13.

It helps repeated keys and version piles on a unique index. It never runs on an `INCLUDE` index, and it does nothing for a key that never repeats.

Covering and dedup pull apart.

Extra `SELECT` columns want `INCLUDE`. Many rows on one key want dedup, which `INCLUDE` disables. Pick the one the workload hits.

Two inserts, two costs.

A new value for a key already in the leaf splits that leaf's `ctids` and leaves the parent alone. A new key copies a separator into the parent, can cascade to the root, and is the costly case under concurrency — worse when the key is unordered.

2.3 Hash Indexes

A hash index answers exact equality without keeping keys sorted. No ranges, no ORDER BY from the index — hash to a bucket, walk that bucket's chain, then confirm the row.

2.3.1 What It Is

Topics so far used the default access method: a **sorted** map from key to ctid. Most indexes you create are that B-tree.

A hash index answers a narrower question: **exact equality** (=), as fast as possible, without keeping keys in order. No ranges, no ORDER BY from the index, no multi-column keys.

```
CREATE INDEX ON users USING hash (email);
-- usable for: WHERE email = 'alice@example.com'
-- useless for: WHERE email LIKE 'alice%'
--              WHERE email > 'm'
--              ORDER BY email
```

Hash Function to Bucket

A heap has no natural address for "alice@example.com". A B-tree finds it by walking sorted fences. A hash index turns the key into a **bucket** number, then reads that bucket's page(s).

```
key → hash(key) → bucket number → bucket page(s)
```

Same key always hashes the same way (**deterministic**). A good hash scatters similar keys (user_001, user_002) into different buckets so one page does not become a hot spot.

In PostgreSQL the index tuple stores the **4-byte hash value** and the `ctid` — not a copy of the column. That keeps the index small for long values (UUID, URL, email). It also means the scan is **lossy**: after the bucket hit, PostgreSQL must still confirm the row matches the real key (heap / recheck). You do not get a covering “key sits in the leaf” story the way a B-tree leaf can.

```
WHERE email = 'alice@example.com'
```

hash(email) → bucket 17

↓

▼

primary bucket page 17

↓ (and any overflow pages chained to it)

▼

candidates: (hash, ctid) ...

↓

▼

heap: confirm email really equals the constant

No tree descent. One meta-driven bucket mapping, then the bucket chain.

Key point

The hash is the address. Everything else in this topic exists because that address is imperfect: collisions, full pages, and growth.

Collisions and Overflow Chains

Many keys, finite buckets — two different keys can share a bucket. That is normal. PostgreSQL does *not* overwrite; it must store both.

Each bucket has a **primary bucket page**. When that page is full, PostgreSQL chains **overflow pages** onto that bucket only — a linked list for the bucket:

```
Bucket 17
  primary page → [ (h1, ctid_A), (h2, ctid_B), ... ]
                  |
                  └─ overflow → [ (h3, ctid_C), ... ]
                        |
                        └─ overflow → ...
```

Lookup: go to the bucket, walk the chain, binary-search *within* each page (entries on a page are sorted by hash code). Across pages in the chain there is no global key order — only “this bucket’s hash values live here.”

Before growing a new overflow page, an insert tries the same cleanup you saw on B-tree leaves: remove index tuples already marked dead, if the page is free enough to do it.

Key point

PostgreSQL hash indexes resolve collisions with **overflow chaining**, not open addressing or probing. Follow the links until the chain ends.

Why a Fat Chain Can Be Worse Than a B-Tree

Cost shape for equality:

AM	Equality cost
Hash	$O(1)$ to the bucket, then walk the <i>whole</i> chain
B-tree	$O(\log n)$ to the first leaf for that key, then only that key's run

Same value many times → same hash → *one* bucket. That chain holds every match. A B-tree also keeps those duplicates together on linked leaves, so “return every match” still touches many pages either way. Hash is not free here.

The case where hash is **clearly worse**: the hash can put **unrelated keys** in the same bucket. An = walk may scan entries that are not your key (and recheck them). A B-tree never does that — internal separators route you *to your key*; if that key has few (or no) matches, you do not pay for other keys that happened to collide.

Hash bucket 17 (skewed)	B-tree leaf run for key K
alice ← your key	... only K's entries ...
carol ← unrelated, same bucket	
dave ← unrelated	
... long overflow ...	

Unique-ish = — the $O(1)$ bucket hit can win. Fat or skewed chains — the “then walk n ” cost dominates, and B-tree’s $\log n$ + only-this-key path wins. Hash works best when values are unique or nearly unique (few rows per bucket).

Growth: Split One Bucket

If the number of buckets were frozen forever, chains would grow without bound. A full rebuild of every entry would fix the bucket count — and would be catastrophic on a large table.

The key→bucket mapping is built so the index can expand **incrementally**. When the index is overfull, PostgreSQL **splits one existing bucket** into two and moves some tuples to the new bucket under the updated mapping. Other buckets stay put.

Before:	After one split:
... bucket 3 (full)	... bucket 3 (kept half)
	... bucket 3+N (new; got the other half)

- No global rehash of the whole index.
- Split work runs in the **foreground** of the insert that triggered it — a table growing very fast can feel that latency.
- Vacuum can later squeeze tuples onto fewer overflow pages and recycle empty overflow pages (tracked via **bitmap pages**). There is no shrink of the bucket count without REINDEX.

That growth pattern is **linear hashing**: expand by splitting one bucket at a time, with overflow chains for local pressure — no full rehash, and no textbook directory-doubling deep dive here.

Page Kinds

Page	Role
Meta (page 0)	Control info: how keys map to buckets
Primary bucket	First page of a bucket
Overflow	Extra pages chained to a full bucket
Bitmap	Tracks freed overflow pages for reuse

2.3.2 Why This Matters

Hash vs B-Tree

	Hash	B-Tree (default)
Operators	= only	=, <, >, BETWEEN, ORDER BY, ...
Columns	Single column only	Multi-column OK
Unique constraint	Not supported	Supported
What is stored	4-byte hash + ctid	Key (+ optional INCLUDE) + ctid
Point lookup	Hash → bucket chain	Root → leaf
Best data shape	Unique / nearly unique equality	General purpose

When hash can win: large table, queries are almost only `col = const`, values are high-cardinality, and you care about a smaller index or skipping a deep B-tree walk. Hash can also take part in bitmap index scans.

When to stay on B-tree: you might ever need a range, a sort, a multi-column key, a unique constraint, or `INCLUDE` / index-only patterns from the previous topic. In practice B-tree is still the right default; hash is a specialist tool.

Modern PostgreSQL hash indexes are **WAL-logged and crash recoverable**. The old “hash indexes aren’t logged” warning is historical. Avoid them for operator and design fit, not durability.

2.3.3 Hands-On Inspection

```
-- equality only, single column
CREATE INDEX users_email_hash ON users USING hash (email);

-- still the usual choice for most workloads
CREATE INDEX users_email_btree ON users (email);
```

Use EXPLAIN on an equality predicate and confirm a hash index scan only appears for `=`. A range or ORDER BY will not use it — create a B-tree if those queries matter.

2.3.4 Summary

At a glance

Equality only.

Hash supports `=` on a single column. No ranges, no ORDER BY from the index, no multi-column keys, no unique constraint.

Hash → bucket → ctid.

The index stores a 4-byte hash and a `ctid`, not the column. Scans are lossy; the real key is confirmed on the heap.

Collisions use chaining.

A full bucket grows a linked list of overflow pages. Lookup walks that chain. PostgreSQL does not overwrite on collision.

Unrelated keys can share a bucket.

That is why a skewed chain can cost more than a B-tree. B-tree separators route you to *your* key; you do not pay for other keys that collided by accident.

Cost shape.

Hash is $O(1)$ to the bucket, then the chain. B-tree is $O(\log n)$ to the leaf, then only that key's run. Unique-ish data favours hash; fat or mixed chains favour B-tree.

Grow by splitting one bucket.

That is **linear hashing** — no full rehash. Split work hits the inserting session. Bucket count does not shrink without REINDEX.

Default stays B-tree.

Reach for hash only when the workload is almost pure high-cardinality equality and you accept the narrower contract.

2.4 BRIN Indexes

A BRIN index does not point at every row. It stores a summary per physical chunk of the heap — typically min and max — so a query can skip page ranges that cannot match. Cheap when data is append-ordered; useless when it is not.

2.4.1 What It Is

B-tree: one entry per row (key → ctid) — exact. Hash: equality → **bucket** → chain.

BRIN (Block Range Index) answers a coarser question: *which ranges of heap pages could contain this value?* It stores a **summary per physical chunk**, not a pointer per row. That only pays off when the column's order on disk matches its logical order — the **correlation** story from the unordered heap earlier in this chapter.

```
CREATE INDEX ON events USING brin (created_at);  
-- strong: append-only time / growing id  
-- weak:  UUID, random keys, heavy updates that scatter old values
```

What BRIN Stores

PostgreSQL stores the heap in 8 KB pages (Chapter 1). BRIN groups consecutive pages into a **block range** and keeps only a summary for the indexed column — typically **min** and **max**.

Default: **pages_per_range** = 128 (≈ 1 MB of heap per index entry).

```

Heap pages 0...127    → min = 100,    max = 500
Heap pages 128...255  → min = 501,    max = 1000
Heap pages 256...383  → min = 1001,   max = 1500

```

	B-tree	BRIN
Entry	One per row (key → ctid)	One per block range (min/max)
Exact?	Yes — jump to specific rows	No — lossy : “matches only in these ranges,” then heap + recheck
Size	Large on big tables	Tiny (often a few MB)
Index-only scan	Possible	Never — always needs the heap

```

WHERE created_at BETWEEN ... -- or =

for each BRIN summary
├ min...max cannot contain value → skip those heap pages
└ might contain value           → keep those pages
    |
    ▼
    bitmap of candidate pages → Bitmap Heap Scan → recheck rows

```

Same lossy idea as hash (confirm on the heap), but the unit is a **page range**, not a bucket of (hash, ctid).

Key point

BRIN does not find the row. It *rules out* ranges that cannot match.

Correlation Decides Everything

BRIN helps only when values that are close **logically** sit close **physically**.

```
SELECT correlation
FROM pg_stats
WHERE tablename = 'events' AND attname = 'created_at';
```

correlation	Meaning	Effect
Near ± 1	Column order matches (or reverse-matches) heap order	Summaries stay tight; many ranges skip
Near 0	Values scattered	Almost every range “matches” $\rightarrow$ BRIN $\approx$ a slow sequential scan

Good: `created_at` / `log_ts` on append-only tables, `bigserial` that only grows at the end.

Bad: random UUIDs, shuffled loads, columns you update often (MVCC places new versions at the end of the heap and widens summaries — below).

The heap is not kept in index order (`CLUSTER` is one-shot). BRIN does not fix layout — it *assumes* inserts already kept the column correlated. A smaller **pages_per_range** will not rescue low correlation.

Why BRIN Wins on Huge Ranges

On a wide predicate (“last 7 days,” tens or hundreds of millions of matching rows), both indexes often read *about the same heap pages*. The difference is the cost of **finding which pages to read**.

- **B-tree**: cheap root→leaf descent, then walk leaves left to right — *one index entry per matching row*. For 100M matches that is **gigabytes of leaf pages** through the buffer cache (and maybe disk), which can **evict** useful pages. Only then the heap.
- **BRIN**: read a **tiny** index (often a few MB, almost always cached), keep overlapping ranges, scan those heap pages sequentially.

B-tree can **pay twice** (huge leaf walk + heap). BRIN **pays once** (tiny summary + heap). BRIN does not bypass the buffer cache — it is just *small enough to stay in it*.

For selective point lookups or narrow ranges, that story flips: B-tree’s exact `ctid` wins, and a cold leaf walk is tiny.

Lifecycle: Summaries, Writes, Fixes

Summarized vs unsummarized.

- **Summarized** — has min/max; can be skipped when it cannot match.
- **Unsummarized** — no entry yet; *always scanned*. Correct, but no skipping.

New pages at the end of an append-only table stay unsummarized until something closes them. That hurts most when the **newest** data is also the **hottest** query (“last 7 days”).

```
SELECT brin_summarize_new_values('events_created_at_brin');
-- or create with: WITH (autosummarize = on)
```

What writes do. Summaries **only widen** — they never shrink on their own.

Operation	Effect
Append INSERT (time / sequence)	Cheap. New tail stays unsummarized until filled or summarized.
Random INSERT	Lands in the wrong physical place → widens that range's min/max. Avoid BRIN.
UPDATE (non-HOT / moves the row)	New heap version often at the <i>end</i> . An old January value can blow open a June range's min.
DELETE	Dead rows leave ghost min/max until the range is re-summarized.

Range widening example:

```
Before: range 50 min = 2026-06-01 max = 2026-06-05
After:  range 50 min = 2026-01-02 max = 2026-06-05
        → January queries must also scan these June pages
```

HOT updates that stay on the same page and do not change indexed columns are kinder (same idea as earlier topics). Anything that moves the indexed value's physical home can widen a summary.

Fixing a ruined range. Vacuum / autovacuum help the unsummarized tail and routine cleanup. A blown-open summary needs an explicit rebuild of that range (or REINDEX):

```
SELECT brin_desummarize_range('events_created_at_brin', block_number);
SELECT brin_summarize_range('events_created_at_brin', block_number);
-- block_number = any page belonging to that range
```

Tuning `pages_per_range`

```
CREATE INDEX events_created_at_brin ON events
  USING brin (created_at)
  WITH (pages_per_range = 32, autosummarize = on);
```

Must be a power of two.

Setting	Trade-off
Smaller (e.g. 32)	Slightly larger index, tighter skipping
Default 128	Fine when correlation is already good
Larger (e.g. 512)	Smaller index, coarser scans

Tune precision only *after* correlation is right.

2.4.2 Why This Matters

Choosing: B-Tree, BRIN, or Both

Prefer BRIN when the table is huge and append-only, the column correlates ($\approx \pm 1$), queries are **wide** ranges, and you care about write cost / index size.

Prefer B-tree when you need selective lookups, unique constraints, `ORDER BY ... LIMIT`, index-only scans, or the data has no correlation — or when the B-tree fits in cache and exactness wins.

Rule of thumb: B-tree for precision; BRIN for huge, ordered, append-only ranges where coarse skipping is enough. Often both:

```
CREATE INDEX events_id_btree ON events (id);  -- point lookups / uniqueness
CREATE INDEX events_created_at_brin ON events
  USING brin (created_at) WITH (autosummarize = on);  -- wide time filters
```

2.4.3 Hands-On Inspection

```
-- check correlation before you commit to BRIN
SELECT correlation
FROM pg_stats
WHERE tablename = 'events' AND attname = 'created_at';

CREATE INDEX events_created_at_brin ON events
    USING brin (created_at) WITH (autosummarize = on);

-- force the hot newest tail if it is still unsummarized
SELECT brin_summarize_new_values('events_created_at_brin');
```

Use EXPLAIN (ANALYZE, BUFFERS) on a wide time range and confirm a Bitmap Index/Heap Scan that skips most of the table. If almost every range survives, correlation is the problem — not **pages_per_range**.

2.4.4 Summary

At a glance

Summary, not per-row.

BRIN stores min/max per block range. Scans are lossy: candidate pages still need a Bitmap Heap Scan and recheck.

Correlation first.

Near ± 1 or skip BRIN. Near 0, almost no pages are skipped.

Huge ranges.

A B-tree may stream gigabytes of leaf pages then the heap. BRIN is a tiny cached map then the heap — it can *pay once* while the B-tree pays twice.

Unsummarized = always scanned.

Hurts most on the newest, hottest tail. Use autosummarize, Vacuum, or brin_summarize_new_values.

Summaries only widen.

Updates and deletes can ruin ranges. Fix with brin_desummarize_range + brin_summarize_range, or REINDEX.

Rule of thumb.

B-tree for precision; BRIN for huge append-ordered ranges. A common shape is both: B-tree on lookup columns, BRIN on time.

2.5 GIN Indexes

GIN flips the index: instead of one key per row, it stores every piece inside the value — key → posting list of ctids. Keys live in a normal B-tree (the entry tree); leaves hold the lists. That is why arrays, jsonb, full text, and trigrams fit here, and why a simple scalar column usually should not.

2.5.1 What It Is

Every index so far answers the same question in a different way: “given a key, find the rows.”

- **B-tree:** one row has one key. Good for equality and ranges.
- **Hash:** one row has one key. Good for equality only, no ranges.
- **BRIN:** many rows share one summary. Good when data is stored in order.

GIN (Generalized Inverted Index) breaks the pattern. Here, **one row can have many keys**. A single `jsonb` document can have ten keys inside it. A single array can have five elements. A single text document can have hundreds of words. A B-tree entry indexes the whole column value once — that is the wrong shape when the value is really a collection of searchable pieces.

GIN flips the index: instead of “row → value,” it stores **key → list of rows**. That is an **inverted index** — the same idea a search engine uses to find which pages contain a word.

```
CREATE INDEX ON articles USING gin (tags);
-- strong: arrays, jsonb, tsvector, trigrams --- many keys per row
-- weak: scalar equality / ranges --- stay on B-tree
```

Inverted Index: Key to Posting List

A book's back-of-book index does not say "page 5 contains these words." It says "word X appears on pages 5, 12, 90." GIN works the same way.

```
Row 1: tags = \{postgres, sql, index\}
Row 2: tags = \{postgres, backend\}
Row 3: tags = \{sql, backend\}
```

A B-tree would need to scan every row to find which rows have `sql`. GIN instead builds this:

Key	Posting List (ctids)
-----	-----
postgres	-> (1, \dots), (2, \dots)
sql	-> (1, \dots), (3, \dots)
index	-> (1, \dots)
backend	-> (2, \dots), (3, \dots)

Each key points to a **posting list**: a list of `ctids` — the same heap item pointers every other access method uses. To find rows where `tags` contains '`sql`', PostgreSQL looks up the key `sql` and reads that list. It does not scan the table for the key lookup itself.

Key point

GIN does not index the value. It indexes every piece inside the value.

That is why GIN fits:

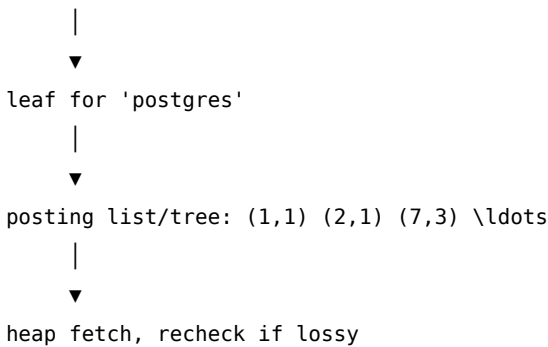
- `integer[]` and other arrays — one key per element
- `jsonb` — one key per top-level key/value pair or path (depends on operator class)
- `tsvector` — one key per lexeme (normalized word)

Under the Hood: Still a B-Tree

GIN does not invent a new tree shape for the key part. The set of distinct keys lives in an **entry tree** — a normal B-tree: same internal nodes, same separators, same descend-left-or-right traversal from Topic 2.1.

The difference is what sits in the **leaf**. In a plain B-tree, a leaf entry points to one heap tuple. In GIN, a leaf entry for a key points to a posting list: all `ctids` of rows that contain that key, stored either **inline** in the leaf (small lists) or in their own small **posting tree** (large lists, keyed by `ctid` instead of by value).

Entry B-tree (keyed by value, e.g. 'postgres')



Lookup path: **B-tree descent to find the key** → read the attached posting list → fetch heap rows → recheck if the opclass is lossy. Same traversal cost you already understand, plus one extra step for however many rows share that key.

Lookup: Extract, Combine, Heap

A GIN query goes through three steps.

Step 1 — Extract keys from the query. If you search tags @> '{"postgres, sql}', PostgreSQL pulls out the keys `postgres` and `sql` from the query value.

Step 2 — Find and combine posting lists. It looks up each key's posting list, then combines them. @> (contains) needs **AND** — a row must appear in both lists. && (overlap) needs **OR** — a row in either list qualifies.

```
postgres -> \{1, 2\}
sql      -> \{1, 3\}

AND (contains both) -> \{1\}
OR  (has either)    -> \{1, 2, 3\}
```

Step 3 — Fetch from heap, recheck if needed. Once GIN has matching ctids, PostgreSQL fetches the rows. For most operator classes this is exact. Some GIN opclasses are **lossy** — they store a compressed signature instead of exact matches (certain `jsonb` path operators). When a match is lossy, the plan shows a **Recheck**: PostgreSQL re-runs the real condition on the fetched row. The index can produce false positives; it must not produce false negatives.

```
EXPLAIN (ANALYZE)
SELECT * FROM articles WHERE body_tags @> '{"postgres"}';

-- Bitmap Heap Scan on articles
--   Recheck Cond: (body_tags @> '{"postgres"}'::jsonb)
--   -> Bitmap Index Scan on idx_gin_tags
```

If you see `Recheck Cond` in the plan, the index used a compressed check first, then verified.

Writes: Pending List and Fastupdate

A single insert into a GIN-indexed column with ten array elements means **ten separate index entries** to update — not one. Batches amplify that into many scattered writes into the index tree.

Why it is worse than it sounds. On a normal B-tree over a time-ordered column (`created_at`), new inserts mostly land in the **rightmost leaf page** — orderly growth, and PostgreSQL can special-case that append pattern. GIN does not get that gift. One row's keys (array elements, `jsonb` paths, lexemes) scatter across the key space — 'apple' and 'zebra' from the same row hit completely different leaves of the entry tree. There is no rightmost page to append to. One insert can touch many random, unrelated leaf pages — expensive if done immediately, entry by entry.

To soften this, GIN uses **fastupdate** (on by default). New entries append first to a small, unsorted **pending list** — a fast append-only buffer — instead of landing immediately in the main GIN structure.

```

Insert --> Pending List (fast, unsorted)
          |
          | (exceeds gin_pending_list_limit, or
          |   autovacuum / manual flush)
          ▼
          Main GIN structure (sorted, searchable by key)

```

Fastupdate ON (default)	Fastupdate OFF
-----	-----
Insert: fast append	Insert: merge into tree now
Lookup: also scan pending	Lookup: always fast
Cleanup: periodic flush	Cleanup: not needed

When the pending list is flushed — by autovacuum, past `gin_pending_list_limit`, or via `gin_clean_pending_list` — PostgreSQL can **sort the whole pending list once** and merge it into the entry tree in one pass. Scattered random writes become fewer, more sequential ones. That is the real win of fastupdate.

Key point

Fastupdate helps bulk inserts, but a long-neglected pending list slows down every read until it is cleaned up.

2.5.2 Why This Matters

Situation	Best	Why
Equality on a unique scalar (<code>id = 5</code>)	B-tree	One key per row; no benefit from splitting
Range queries (<code>created_at > ...</code>)	B-tree	GIN has no natural ordering across keys
Array contains / overlap	GIN	Many keys per value; needs inverted lookup
<code>jsonb</code> key/path containment	GIN	Same — many keys per document
Full text (<code>tsvector @@ query</code>)	GIN	Many lexemes per document
Heavy write on the indexed column	B-tree (or tuned GIN)	GIN write amplification is real

Do not reach for GIN by default. If the column holds one simple value per row and you only check equality or ranges, a B-tree is simpler, smaller, and cheaper to write. GIN earns its cost when a single row's value is really a *collection* of searchable pieces.

Common Shapes

A quick look at what each opclass is *for* — not a full operator list. Every example below is the same pattern: one column, many searchable keys inside it; index the keys instead of the whole value.

Arrays — membership and overlap:

```
CREATE INDEX idx_gin_tags ON articles USING gin (tags);

SELECT * FROM articles WHERE tags @> '{postgres}'; -- contains
SELECT * FROM articles WHERE tags && '{sql,index}'; -- overlaps
```

`jsonb` — key/value or path containment:

```
CREATE INDEX idx_gin_data ON articles USING gin (data);

SELECT * FROM articles WHERE data @> '{"status": "published"}';
```

tsvector — full text search:

```
CREATE INDEX idx_gin_fts ON articles
  USING gin (to_tsvector('english', body));

SELECT * FROM articles
WHERE to_tsvector('english', body) @@ to_tsquery('postgres & index');
```

Trigrams (pg_trgm) — fuzzy text and substring match. The extension breaks text into overlapping 3-character chunks (**trigrams**), then indexes them the inverted way: trigram → posting list of ctids. Example: "postgres" → " p", " po", "pos", ...

```
CREATE EXTENSION pg_trgm;
CREATE INDEX idx_trgm_gin ON articles
  USING gin (title gin_trgm_ops);

SELECT * FROM articles WHERE title LIKE '%postgres%'; -- mid-string
SELECT * FROM articles WHERE title % 'postgress';     -- fuzzy / typo
```

That is what lets LIKE '%term%' use an index at all — a B-tree cannot help when the match is not anchored at the start. pg_trgm can also build on **GiST** (gist_trgm_ops): GIN tends to give faster lookups and a larger index; GiST builds and updates faster but reads a bit slower.

Key point

GIN does not care what the key is — array element, jsonb path, lexeme, or trigram. It only cares that a key can point to many rows.

GiST can index some of the same types with a different internal structure and different trade-offs — see Topic 2.6. GIN remains the common default for read-heavy inverted cases; GiST is often preferred when build and update cost dominate.

2.5.3 Hands-On Inspection

```
CREATE INDEX idx_gin_tags ON articles USING gin (tags);

EXPLAIN (ANALYZE, BUFFERS)
SELECT * FROM articles WHERE tags @> '{postgres}';

-- if inserts feel fast but lookups drag after bulk load:
SELECT gin_clean_pending_list('idx_gin_tags');
```

Confirm a Bitmap Index Scan on the GIN index for containment. After a large load with fastupdate on, clean the pending list (or wait for autovacuum) and re-check buffers / timing.

2.5.4 Summary

At a glance

Inverted index.

Stores key → posting list of ctids, not row → whole value.

Entry tree.

Keys live in a normal B-tree; leaves hold posting lists (or posting trees for large lists).

Lookup.

Extract keys from the query → combine posting lists (AND/OR) → fetch heap rows; recheck when the opclass is lossy.

Pending list / fastupdate.

New entries land in a fast unsorted buffer first. Flush sorts then merges (`gin_pending_list_limit`, Vacuum, or `gin_clean_pending_list`).

Write cost.

Many keys per row → random leaf hits (no rightmost-append). Heavier than a B-tree on a scalar.

Best fit.

Arrays, jsonb, full text, trigram fuzzy / substring search — many keys per row.

Avoid.

Simple scalar equality or ranges — use a B-tree instead.

2.6 GiST Indexes

GiST is a balanced tree of summaries, not a single sortable key. Postgres owns the tree machinery; an operator class plugs in contains / overlaps / distance for ranges, points, and shapes. Light on purpose: the decision surface, not every opclass internals dump.

2.6.1 What It Is

GiST (Generalized Search Tree) is, like GIN, a **framework** rather than one index for one type. Postgres owns the tree machinery — balancing, page splits, traversal. An **operator class** plugs in the logic for a specific type: how to compare two values, how to summarize a group of them, how to decide if a branch might contain a match.

The sentence to hold onto: GiST is a **balanced tree where every node stores a summary of what is below it**, instead of a single sortable key. The rest of this topic is why that matters and what it costs.

The Problem: Columns That Are Not One Sortable Value

Every index so far (B-tree, hash, BRIN, GIN) assumes you can eventually reduce things to values on **one sorted line** — numbers, text, or individual keys pulled out of a bigger value.

GiST exists for columns where the value itself is a **shape**: a **range** (start and end), a **point** (x and y together), a box, a polygon. These still live in a single column — `tsrange`, `point`, `box` are real Postgres types — but the value inside has structure and **width**, not just one comparable position.

```
CREATE TABLE rooms (  
    room_id int,  
    start_date date,  
    end_date date  
);  
  
-- overlap find (works, but indexing and concurrency are awkward):  
SELECT * FROM rooms  
WHERE room_id = 101  
    AND start_date < :new_end  
    AND end_date > :new_start;
```

That query runs, with two real problems:

- **Indexing gets awkward.** A B-tree accelerates one inequality well. Here you have two independent inequalities on two columns. Postgres can use an index for one side; the other becomes a row-by-row filter — not a clean single tree descent. As the table grows, this drifts toward scanning a wide slice, and in the worst case toward a sequential scan.
- **It cannot safely prevent overlaps.** Check-then-insert is not a constraint: two transactions can both check before either commits, both see no conflict, and both insert — leaving an overlapping booking.

A single `tsrange` column, plus an index type that natively understands overlap, solves both. That is what GiST is for.

B-Tree Separator vs GiST Summary

B-tree: the separator is a rule decided *in advance*, and it is exact:

```
Separator = 50
Left:  ALL keys < 50    (guaranteed, no exceptions)
Right: ALL keys >= 50   (guaranteed, no exceptions)
```

Because every key is a single point, there is no in-between case. A query only ever needs to follow **one path** down the tree.

GiST: there is no cutoff decided in advance. Each internal node looks at whatever values already ended up below it and computes: *what is the smallest range/box that covers everything down there?* That summary is computed from the data, not a rule imposed on it.

Because ranges have width, two separate summaries can physically overlap. Walk through numbers:

```
Left child holds:  [1,3]  and  [2,10]
Right child holds: [9,15] and [20,25]

Left summary  = cover of [1,3] and [2,10]  -> [1, 10]
Right summary = cover of [9,15] and [20,25] -> [9, 25]
```

[1,10] and [9,25] overlap between 9 and 10 — because [2,10] is wide enough to stretch the left child's coverage into territory the right child also covers. That is not a “bad” split; it is an unavoidable consequence of one range being large. Even a well-organized split can still leave overlapping summaries if any single range inside a group is wide.

Key point

B-tree separators are exact cutoffs decided up front — one path. GiST summaries are descriptions computed from the data; width can make them overlap — more than one path may be needed.

Traversal: Overlapping Summaries Mean Extra Branches

Search for anything overlapping day 9.5 in the tree above:

```
Is 9.5 possibly inside left coverage [1, 10]? -> yes, maybe
Is 9.5 possibly inside right coverage [9, 25]? -> yes, maybe
```

```

      [Left: covers 1-10]      [Right: covers 9-25]
        \                      /
          both say "maybe" for 9.5
        /                      \
  check leaf pages      check leaf pages
    under left          under right

```

Both answers are “maybe,” so Postgres descends into **both** subtrees and checks the leaf pages under each. A B-tree never does this: a single point value cannot straddle an exact cutoff. Here it happens because the summaries themselves overlap in space.

This is a pure **traversal-cost** issue — how many leaf pages you visit. It has nothing to do with whether the stored value is exact or compressed. That is a separate concern.

A Separate Issue: Lossy Storage and Recheck

Some GiST opclasses **compress** the value to keep the index small — a full polygon reduced to a cheap bounding box, for example. That compressed summary is fast to compare, but not always exact: it can say “maybe” when the real shape does not match.

Postgres adds a **Recheck** step: re-test the real, uncompressed row before returning it, discarding false positives.

```
EXPLAIN (ANALYZE)
SELECT * FROM rooms WHERE during && tsrange('2026-01-01', '2026-01-05');

-- Bitmap Heap Scan on rooms
--   Recheck Cond: (during && '[...]'::tsrange)
--   -> Bitmap Index Scan on idx_gist_during
```

Keep the two causes separate — they are not the same thing:

Cause	Effect	Trigger
Overlapping summaries	Visit more than one leaf page	Data has width (e.g. a wide range)
Lossy / compressed storage	Recheck the real row after a match	The opclass compresses values to save space

You can have either without the other: two leaf pages checked with no recheck (exact types like `tsrange` usually are not compressed), or exactly one leaf page checked but a recheck still required (a lossy geometry opclass). B-tree has neither problem: comparisons are always exact, and each key maps to exactly one path.

2.6.2 Why This Matters

Situation	Best	Why
Equality / range on a scalar	B-tree	Exact ordering; single path
Array / jsonb / FTS (read-heavy)	GIN	Many discrete keys per row; faster lookups
Same, but insert-heavy / build time	GiST	Faster to build and update; slightly slower reads
Range overlap (tsrange, ...)	GiST	Overlap is not equality — needs summaries
Prevent overlapping rows (no double-book)	GiST + EXCLUDE	A constraint, safe under concurrency
Geometric / PostGIS shapes	GiST	No single sortable order for shapes
Nearest neighbors (KNN)	GiST	Native ordered distance scan; GIN cannot

2.6.3 Hands-On Inspection

Prevent double-booking a room:

```
CREATE TABLE rooms (
  id serial PRIMARY KEY,
  room_id int,
  during tsrange
);

CREATE INDEX idx_gist_during ON rooms USING gist (during);

ALTER TABLE rooms ADD CONSTRAINT no_overlap
  EXCLUDE USING gist (room_id WITH =, during WITH &&);
```

Read it as: for the same `room_id`, no two rows may have overlapping `during` ranges. Unlike a manual check-then-insert, this is enforced by the database and is safe if two inserts race.

```
INSERT INTO rooms (room_id, during)
VALUES (101, '[2026-01-03, 2026-01-06]');
INSERT INTO rooms (room_id, during)
VALUES (101, '[2026-01-05, 2026-01-08]');
-- ERROR: conflicting key value violates exclusion
--       constraint "no_overlap"
```

Nearest neighbors to a point:

```
CREATE INDEX idx_gist_location ON places USING gist (location);

SELECT name FROM places
ORDER BY location <-> point '(0,0)'
LIMIT 5;
```

`<->` means “distance to.” A B-tree can sort one column, but there is no single ascending order for “how close is this 2D point to that one.” GiST’s spatial tree can walk outward toward the nearest matches directly.

Geometric / PostGIS:

```
CREATE INDEX idx_gist_geom ON parcels USING gist (geom);

SELECT * FROM parcels WHERE geom && ST_MakeEnvelope(0,0,10,10);
```

Trigram / full text (alternative to GIN):

```
CREATE INDEX idx_gist_trgm ON articles
  USING gist (title gist_trgm_ops);
```

Faster to build and update than the GIN version, a bit slower to read — pick based on whether the workload is write-heavy or read-heavy (Topic 2.5).

2.6.4 Summary

At a glance

Shape columns, not scalars.

Ranges, points, boxes, polygons have width; B-tree's single-line ordering does not apply cleanly.

Framework + opclass.

Postgres owns the tree; the operator class plugs in compare / summarize / "might contain."

Summaries, not separators.

B-tree cutoffs are exact and pre-decided. GiST summaries are computed from data below; width can make sibling summaries overlap.

Multi-branch traversal.

Overlapping summaries are a traversal-cost issue — more than one leaf path may be visited.

Recheck is separate.

Lossy compression can produce false positives; Postgres rechecks the real row. Overlap and lossiness are independent causes.

EXCLUDE for safety.

EXCLUDE USING gist (... WITH &&) stops overlapping rows under concurrency — a constraint, not a check-then-insert race.

KNN is a GiST strength.

ORDER BY col <-> point LIMIT n walks distance order from the index; GIN has no equivalent.

Rule of thumb.

If the question is overlaps / contains / nearest on a range, point, or shape — reach for GiST.

2.7 Choosing an Index

Five access methods, one question: how do you actually access this column? This topic is not new machinery — it is the decision framework that ties B-tree, hash, BRIN, GIN, and GiST together.

2.7.1 What It Is

You have now seen five index types: B-tree, hash, BRIN, GIN, GiST. Each exists because of a different answer to the same question: **how do you actually access this column?** This topic is not new material — it is a decision framework for everything already covered in this chapter.

Ask the Column Three Questions

Before picking an index, ask:

Scalar or not? Scalar means one plain comparable thing per row — a number, a date, a short string. Not scalar means the value itself has internal structure: a range with a start and end, a point with x/y , a document with many keys inside it.

Trap

“Scalar or not” is not a property of the column’s declared type — it is a property of how you choose to access it. A `text` column is a scalar type. Queried with `WHERE title = 'exact string'`, you treat it as one comparable unit and B-tree fits. Queried with fuzzy or substring matching (`pg_trgm`, `LIKE '%...%'`), you stop treating that same string as one unit — you ask “does this string contain this chunk,” the same question GIN answers for

`jsonb` keys or array elements. Same column, same bytes on disk, two different index decisions, because the access pattern changed, not the data type.

If it is not scalar *by access pattern*: discrete or continuous? Discrete is a bag of separate, independent items with no relationship to each other — array elements, `jsonb` keys, words, trigrams: each is present or not, with no overlap between them. Continuous means the value has width or position, and two values can partially overlap or sit at a distance — ranges, points, boxes.

What is the actual operation? Equality only? A range or inequality? “Contains this item”? “Overlaps this span”? “Nearest to this point”? “This column is huge and I mostly filter on order, not lookup”?

The Decision Table

Column shape	Access pattern	Index
Scalar	Equality only	Hash
Scalar	Equality or range / ordering	B-tree
Scalar, correlated with physical row order (time-stamps, IDs)	Range filter on a huge, append-mostly table	BRIN
Not scalar, discrete (array, <code>jsonb</code> , words, trigrams)	“Does this contain / include this key?”	GIN
Not scalar, continuous (range, point, box, polygon)	“Does this overlap / contain / lie near that?”	GiST

Answer the three questions and you can find this row without memorizing anything — the shape of the data and the shape of the question determine the index, not the other way around.

Worked Examples

<code>user_id = 42</code>	-> scalar, equality only	-> Hash (or B-tree)
<code>created_at > '2026-01-01'</code>	-> scalar, range	-> B-tree
<code>created_at > '2026-01-01'</code>	-> scalar, huge append-only log	-> BRIN
<code>tags @> '{postgres}'</code>	-> discrete collection	-> GIN
<code>data @> '{"status":"live"}'</code>	-> discrete collection (jsonb)	-> GIN
<code>title LIKE '%postgres%'</code>	-> discrete (trigrams)	-> GIN or GiST
<code>during && tsrange(...)</code>	-> continuous, overlap	-> GiST
<code>location <-> point '(0,0)'</code>	-> continuous, distance	-> GiST
<code>EXCLUDE USING (... WITH &&)</code>	-> continuous, overlap constraint	-> GiST

`title LIKE '%postgres%'` can go either way — that is not a contradiction. It is the point from the trap above: the same `text` column is scalar when queried by equality, and effectively a discrete collection of trigrams when queried by fuzzy match. The access pattern decided the shape, not the column's type. The next subsection explains why GIN and GiST can both host that same trigram logic.

2.7.2 Why This Matters

GIN and GiST Are Frameworks, Not Fixed Indexes

GIN and GiST are not one algorithm each — they are **hosts**. Postgres builds the tree machinery once per framework: page layout, balancing, splitting, WAL logging, locking, traversal. Neither framework knows anything about `jsonb`, arrays, ranges, or geometry on its own. What it knows is a small set of questions it will ask, and it expects an **operator class** to answer them.

For GIN, the host asks the opclass:

- “Break this row’s value into its keys” (`extractValue`)
- “Break the query into keys to look up” (`extractQuery`)
- “Given which keys matched, does this row actually satisfy the condition?” (`consistent`)

For GiST, the host asks the opclass:

- “Could a match exist under this summary?” (`consistent`)
- “Merge these child summaries into one parent summary” (`union`)
- “This page is full — how do you divide these entries into two?” (`picksplit`)
- “How far is this value from the query?” (`distance`, for KNN)

An operator class is a plug-in that answers these questions for one data type. It does not write tree code, manage pages, or handle concurrency — only the type-specific logic. Built-in opclasses (`jsonb_ops`, range types, `tsvector`) are exactly that: plug-ins shipped with core.

That is exactly what `pg_trgm` is. The extension does not add a new index type — it registers two new operator classes, `gin_trgm_ops` and `gist_trgm_ops`, that teach the existing GIN and GiST hosts how to break a string into trigrams and how to compare them. Same tree engine, same page structure, same traversal code — only the key-extraction and comparison logic changed.

```

      GIN engine (tree, pages, traversal --- fixed)
      /           |           \
  jsonb_ops      default tsvector ops  gin_trgm_ops (pg_trgm)
  "keys = jsonb"  "keys = lexemes"      "keys = 3-char chunks"
  paths"

```

```

      GiST engine (tree, pages, traversal --- fixed)
      /           |           \
  range opclasses  geometry opclasses  gist_trgm_ops (pg_trgm)
  "summary = range" "summary = bounding" "summary = trigram
  union"           box"                set similarity"

```

Key point

Trigram search can live under either GIN or GiST because the data (trigrams) is the same. What changes is which host you plug that opclass into — engine trade-off (GIN: faster reads, slower writes; GiST: faster writes, slower reads), separate from what the keys mean.

2.7.3 Summary

At a glance

Access pattern first.

“Scalar or not” is how you query the column, not only its declared type — equality on text is scalar; trigram LIKE on the same column is discrete.

Three questions.

Scalar or not? If not, discrete or continuous? Then: which operation — equality, range, contains, overlap, nearest?

Decision follows shape.

Match column shape + access pattern to the AM built for it — hash, B-tree, BRIN, GIN, or GiST.

GIN and GiST are hosts.

Fixed tree / page / traversal machinery; an operator class supplies type-specific logic.

Operator class = plug-in.

Answers extract / consistent / union / picksplit / distance — never owns tree internals directly.

Extensions add opclasses.

pg_trgm registers gin_trgm_ops and gist_trgm_ops; same engines, new key logic.

The real skill.

Not memorizing five index types — reading a column's shape and access pattern, then picking the AM built for that shape.

2.8 Partial Indexes

Choosing an index answered which access method. This topic is a different cut: which rows get into the index at all. Same tree, same leaf layout — only a subset of the table participates.

2.8.1 What It Is

A **partial index** is an index with a `WHERE` predicate on the definition. Only rows that satisfy that predicate get an entry. Everything else is simply absent from the index. Same B-tree machinery already covered in this chapter — the difference is **membership**: the tree only contains the hot slice.

```
CREATE INDEX ON orders (created_at)
  WHERE status = 'pending';
```

SQL Server calls the same idea a *filtered index*. The Postgres name is partial; the meaning is identical.

2.8.2 Why This Matters

Key point

A partial index is not a new access method. It is a membership filter on an existing one — usually B-tree — so only matching rows pay the storage and write cost.

A partial index rarely makes the hot-path read *much* faster than a well-designed full index on the same slice. Its value sits on the **cost side** — and in one place a full index cannot go at all.

Smaller Index on a Skewed Table

Many tables have a small hot slice and a huge cold majority: pending orders against millions of completed ones, queued jobs against historical done, active users against soft-deleted rows, unread notifications against read ones.

A full index stores every row. A partial index stores only the slice the application actually looks up. That means less disk, a better chance the whole index stays in shared buffers, a shallower tree, and less work for `VACUUM` and `REINDEX`.

Cheaper Writes Outside the Slice

Every insert, update, and delete must maintain each index that applies to the row. With a partial index, rows that never match the predicate **never touch the index**. Less write amplification, less WAL. When a row leaves the predicate — pending to completed — Postgres simply drops its entry. That is the intended lifecycle for a work-queue row.

Conditional Uniqueness

This is the one thing a full unique index cannot express: uniqueness that applies only to some rows.

```
CREATE UNIQUE INDEX ON users (email)
WHERE deleted_at IS NULL;
```

Two soft-deleted users may share an email with a new active account. Two *active* users cannot. The same pattern covers “only one primary address per user” (`WHERE is_primary`) and similar business rules. A check-then-insert in application code is not a substitute — concurrency will race you. The partial unique index is the constraint.

What It Does Not Fix: Selectivity

Trap

“Status is low-cardinality, so I need a partial index for the rare values.” That is wrong in Postgres. The planner already has per-value frequencies in `pg_stats` (`most_common_vals / most_common_freqs`). On a full index over `status`, a rare value at roughly 0.5% of the table is estimated as selective and uses the index; a dominant value at 99% correctly falls back to a sequential scan. A full index already handles rare lookups. Partial does not make the planner smarter about selectivity — it makes the index cheaper to store and maintain, and it unlocks conditional uniqueness.

Estimates can still go wrong for the usual reasons: stale statistics after a big data shift, values outside the MCV list (fallback via `n_distinct`), or a generic prepared plan where the parameter value is unknown at plan time. Those are statistics and planning problems, not reasons to reach for a partial index as a selectivity fix.

Full Index vs Partial Index

Take the same hot-path query:

```
SELECT * FROM orders
WHERE status = 'pending'
ORDER BY created_at
LIMIT 50;
```

Two designs answer it:

Design	Definition (conceptually)
Full composite	Index (status, created_at) over every row
Partial	Index (created_at) only where status = 'pending'

On the read, both are about the same. In the full composite, status is the leading key, so all pending entries sit in one contiguous range — Postgres jumps there and walks fifty ordered rows without reading the completed mass. The partial index is simply smaller: it never stored that mass at all.

Goal	Full	Partial
Fast read of the rare slice	yes	yes (about the same)
Small index / cheap cold writes	no	yes
Conditional uniqueness	impossible	yes

At scale the size gap is the story. On a 50M-row table with roughly 0.5% pending, the full index holds about 50M entries; the partial holds about 250K — roughly two hundred times smaller, no index maintenance for non-pending writes, and comparable read speed for the pending queue.

Minor read advantages that do exist: better cache locality when the whole index fits in memory, a slightly shallower tree, fewer pages for index-only scans and vacuum. Real, usually secondary to the write and size wins.

2.8.3 Hands-On Inspection

When the Planner Will Use It

Postgres uses a partial index only when it can **prove** that the query's `WHERE` clause implies the index predicate.

- Query includes `status = 'pending'` (or an equivalent the planner recognizes) — can use the index.
- Query filters only on `created_at` with no status predicate — cannot use it; the index does not claim to cover those rows.
- Query uses `WHERE status = $1` under a generic prepared plan — the planner cannot assume `$1` is 'pending', so it may refuse the partial index even when the runtime value would match.

Trap

ORMs and prepared statements love parameters. If the partial index is the point of the design, the hot path usually needs the constant in the SQL (or a custom plan path), not a bind parameter for the predicate value. Small rewrites of the predicate can also break the implication proof — confirm with `EXPLAIN (ANALYZE, BUFFERS)`.

Patterns Worth Knowing

Pattern	Predicate idea	Indexed key
Work queue	<code>status = 'queued'</code>	<code>(priority, created_at)</code>
Soft delete	<code>deleted_at IS NULL</code>	columns you look up for live rows
Sparse non-NULL	<code>refund_id IS NOT NULL</code>	<code>(refund_id)</code> — skip the NULL majority
Unread / unprocessed	<code>read_at IS NULL</code>	<code>(user_id, created_at)</code>
Hot path + covering	same hot predicate	key columns + <code>INCLUDE (...)</code>

The soft-delete and sparse-NULL cases are easy to under-appreciate: a full index would spend most of its leaf space on rows you never seek.

When to Reach for One

Use a partial index when the data is skewed (small hot slice, large cold majority), the hot queries consistently filter that slice, and the table is large enough that index size and write cost matter — **or** when you need conditional uniqueness, which is a strong reason at any size.

Skip it when the table is small, values are evenly used, queries span the whole domain, the predicate churns often, or you cannot control parameterization on the hot path.

2.8.4 Summary

At a glance

Membership, not a new AM.

Same tree; fewer rows participate. The `WHERE` on `CREATE INDEX` is a filter on who gets an entry.

Cost, not selectivity.

Smaller index and cheaper cold writes. The planner already uses per-value stats on a full index for rare values.

Conditional uniqueness.

The one capability a full unique index cannot express — uniqueness that applies only to some rows.

Implication proof.

The query `WHERE` must imply the index predicate, or the planner ignores the partial index.

Parameters are a trap.

Generic plans with `$1` often cannot assume the predicate value; prefer a constant on the hot path.

Design test.

Skewed hot slice you always filter on — or a uniqueness rule that only applies to some rows.

2.9 Composite Indexes

Partial indexes answered which rows get an entry. This topic is a different cut: how many columns share one key. Same B-tree — one sorted sequence — but the key is a tuple, and that order decides which queries can use the tree.

2.9.1 What It Is

A **composite index** (also *compound* or *multi-column*) is an index on two or more columns. The leaf does not hold each column as its own tree. It holds one concatenated key, sorted in declaration order.

```
CREATE INDEX ON events (tenant_id, event_type);
```

Entries are ordered first by `tenant_id`, then by `event_type` within each tenant. The engine seeks that combined order — not two independent indexes glued together.

2.9.2 Why This Matters

Key point

When a query filters on multiple columns, one single-column index can narrow rows directly; the second predicate is reconstructed at runtime — as a `Filter`, or as a bitmap merge — unless both columns share one key.

The Problem

That is a structural limit. Given two separate single-column indexes, the planner has essentially two strategies, and both are compromises.

Strategy 1: Pick one index, filter the rest in memory. The planner estimates which column is more selective — from row counts, value distribution, histograms in `pg_stats` — and uses that index to fetch candidates. The second condition is then evaluated **row by row** as a `Filter` after the Index Scan. Every row that matched the first condition and failed the second was still fetched, held, and checked — wasted work. And because the choice rests on estimates, not certainty, stale statistics, correlated columns, or skew can make the planner pick the *worse* of the two indexes.

Strategy 2: Use both indexes via a bitmap scan, and merge. Instead of committing to one index, the planner scans both, builds an in-memory **bitmap** of matching rows (or pages) for each, and ANDs (or ORs) the bitmaps before fetching only survivors. That avoids pulling the full unfiltered set from either side — but it pays for two index probes, a bitmap merge, a recheck against the heap, and, if the result is large, a **lossy** bitmap once it exceeds `work_mem` (pages instead of exact TIDs), which forces still more rechecking.

In both strategies the issue is the same: **at query time** the engine is reconstructing a relationship between two columns that could have been built once, ahead of time, into the index. Neither single-column index knows about the other, so every execution pays CPU for filtering, or CPU and RAM for bitmaps, to compensate.

A composite index fixes that at the root. It stores both columns in one sorted structure, so the intersection is one direct lookup — no in-memory filter for the second predicate, no bitmap merge, no dependence on which single-column index looked “more selective.”

A Concrete Example

Multi-tenant tables make the abstract problem concrete. Nearly every query leads with `tenant_id`:

```
SELECT * FROM events
WHERE tenant_id = 42 AND event_type = 'login';
```

`tenant_id` alone is poorly selective — a large tenant may own millions of rows. `event_type` may be more selective, but still common enough across *all* tenants that it does not pin the result by itself. With two single-column indexes, the planner faces exactly the two strategies above:

```

-- Strategy 1: pick one index, filter the rest in memory
Index Scan using idx_event_type on events
  Index Cond: (event_type = 'login')
  Filter: (tenant_id = 42)          -- row by row, after the lookup

-- Strategy 2: bitmap AND of both indexes
Bitmap Heap Scan on events
  Recheck Cond: (tenant_id = 42 AND event_type = 'login')
  -> BitmapAnd
    -> Bitmap Index Scan on idx_tenant_id
      Index Cond: (tenant_id = 42)
    -> Bitmap Index Scan on idx_event_type
      Index Cond: (event_type = 'login')

```

With the composite, both conditions sit in Index Cond:

```

Index Scan using events_tenant_id_event_type_idx on events
  Index Cond: (tenant_id = 42 AND event_type = 'login')

```

The B-tree is sorted first by `tenant_id`, then by `event_type` within each tenant. The engine jumps to the (42, 'login') slice — one lookup, no Filter, no bitmap merge, no recheck for those predicates.

Why the Composite Wins Over Both Strategies

Approach	What you pay beyond the matching slice	Selectivity bet?
Single index + Filter	All candidates of the chosen index, then a row-by-row filter; candidates held in memory while filtering	Yes — wrong pick is costly
Two indexes + BitmapAnd	Both indexes scanned; bitmap build, merge, and recheck. Bitmaps live in <code>work_mem</code> and can go lossy	No, but fixed overhead always
Composite index	Only the matching leaf slice; one traversal. No intermediate structure	No — the key encodes the match

The composite does not merely pick the better workaround. It **removes the need for either one**: the relationship between the columns is encoded at index-build time, not recomputed on every execution.

The Trade-off Up Front

That efficiency is not free. A composite is larger and costs more on every write than a narrow single-column index. And it is less reusable: `(tenant_id, event_type)` does not help a query that filters only on `event_type`. That limit is the **leftmost prefix rule** — below. The bargain is deliberate: more storage and write cost, in exchange for deleting runtime filter and merge work from the hot path.

How the Key Is Ordered

Think of each leaf entry as a tuple:

tenant_id	event_type	Index key
3	login	(3, login)
3	logout	(3, logout)
42	login	(42, login)
42	purchase	(42, purchase)

The B-tree compares keys lexicographically: first column, then second, then third. **Column order is the design**, not a cosmetic detail. `(tenant_id, event_type)` and `(event_type, tenant_id)` are different indexes. They serve different query shapes.

The Leftmost Prefix Rule

An index on `(A, B, C)` can efficiently support predicates that pin a **leftmost prefix** of the key:

- `A`
- `A and B`
- `A and B and C`

It cannot efficiently support predicates that skip the leading column: only `B`, only `C`, or `B and C` without `A`.

The tree is sorted by `A` first. Without a constraint on `A`, values of `B` are scattered across the whole leaf chain. The engine would have to walk the entire index — at which point a sequential scan or a dedicated index on `B` is the honest answer.

```
-- Index: (tenant_id, event_type)

-- Uses the index: leading column is constrained
WHERE tenant_id = 42
WHERE tenant_id = 42 AND event_type = 'login'

-- Does not use it efficiently: leading column missing
WHERE event_type = 'login'
```

Trap

You bought the `(tenant_id, event_type)` seek, so a global “all logins” query needs its own index (or a different leading column). Postgres may still touch a composite for a non-leading column via an index-only scan that filters as it walks — but that is not the design win. Design for the leftmost-prefix shape you actually query.

Column Order: What Determines “Right”

There is no universal best order for three columns. Order follows the query mix.

Equality before range. Columns used with `=` should lead columns used with `>`, `<`, or `BETWEEN`. After the first range predicate, later columns no longer form a single contiguous seek range for further filtering — they can still help as a filter while walking, or as `INCLUDE` payload, but the sorted-range story stops at the first inequality.

So for `WHERE tenant_id = 42 AND created_at BETWEEN ...`, put `tenant_id` first. Putting the range column first wastes the equality column’s ability to collapse the tree to one slice.

Match the hot path, not abstract selectivity. In the multi-tenant case, `tenant_id` often leads *even though it is low-selectivity alone*, because every query starts there — and the second column supplies the selectivity inside that tenant. A rarely filtered ultra-selective column is a poor leading key if most lookups never mention it. The leftmost prefix you pay for should be the leftmost prefix you query.

One composite per dominant shape. `(tenant_id, event_type)` and `(event_type, tenant_id)` serve different applications. If two shapes are both hot and neither shares a prefix, you need two indexes — the leftmost rule will not stretch one tree to cover both.

2.9.3 Hands-On Inspection

Filter Plus Sort

The quiet second win is skipping Sort — same leftmost idea, different second column:

```
WHERE tenant_id = 42
ORDER BY created_at
LIMIT 50;
```

With `(tenant_id, created_at)`, Postgres seeks to tenant 42 and walks fifty already-ordered leaf entries. With only `(tenant_id)`, it finds the tenant's rows and then sorts. With only `(created_at)`, it cannot pin the tenant without filtering. The composite turns filter + order into one ordered range scan.

This is the same reason covering and partial designs often end with a time or priority column: the last key column is there so the walk *is* the sort.

Covering: Already Covered

If every column the query needs sits in the index — keys or `INCLUDE` — the plan can become an index-only scan. That mechanism was Section 2.2. A composite often becomes covering by accident when you add the columns you filter and sort on; reach for `INCLUDE` when you want payload without changing key order or leftmost-prefix behavior.

Do not widen the *key* just to cover. Extra key columns change which prefixes the index serves and inflate every comparison. Payload that is only selected belongs in `INCLUDE`.

Trade-offs

Consideration	What it means
Write cost	Every insert/update/delete maintains the wider key
Storage	More columns per leaf entry → larger index
Leftmost lock-in	(A, B) does not help queries that only filter on B
Wide keys	Beyond two or three key columns, reasoning and reuse collapse fast
Overlap with singles	A good composite often makes a leading-column single-column index redundant — keep the extra index only if a non-prefix shape needs it

When to Reach for One

Use a composite when hot queries always lead with the same column (tenant, customer, account) and then constrain or sort on a second — especially when the leading column alone is too coarse to be selective. That shape is stable enough to bake into key order.

Skip a wide composite when each column is queried alone in different shapes, when a single-column index already answers a selective lead, or when you are tempted to index “every column the SELECT list mentions” as keys instead of using INCLUDE.

2.9.4 Summary

At a glance

Structural gap.

Two single-column indexes force a runtime `Filter` or `BitmapAnd`. The engine reconstructs a relationship the index never stored.

Combined key.

Intersection resolved in the tree — encoded once at build time, not recomputed per query.

Leftmost prefix.

(A, B, C) serves A / A,B / A,B,C — not B alone.

Equality before range.

The first inequality ends the contiguous seek story for later key columns.

Filter + sort.

Leading equality plus next-column ORDER BY can skip Sort entirely.

Cover with INCLUDE.

Payload is not a key; do not widen the prefix for SELECT-only columns.

Design test.

Stable leading column (often low-selectivity alone) plus a second column that finishes the lookup.

2.10 Expression Indexes

Composite indexes answered how many columns share one key. This topic is a different cut: what value is stored as the key. Same B-tree — but the leaf holds the result of an expression, not the raw column bytes.

2.10.1 What It Is

An **expression index** (also *functional index*) is an index whose key is an expression in parentheses, not a bare column list.

```
CREATE INDEX ON users ((lower(email)));
```

The double parentheses are required: the outer ones are the index column list; the inner ones mark an expression. Postgres stores `lower(email)` for each row. Lookups that use the same expression can seek that tree.

Same access methods as elsewhere in this chapter — usually B-tree. The novelty is not a new AM. It is **what** gets indexed.

2.10.2 Why This Matters

Key point

A normal index stores column values as they sit on the heap. If the query asks for a derived form of that column, the planner has no sorted key for the value it needs — unless the index itself stores that derivation.

The Problem

A normal B-tree index stores column values as they sit on the heap. The planner can seek that tree only when the query predicate lines up with those stored values — equality or range on the column itself.

Real queries often do not ask about the column as stored. They ask about a **derived** form of it:

- case-insensitive match — `lower(email)`
- calendar day of a timestamp — `(created_at::date)` or `date_trunc('day', created_at)`
- one field from JSON — `(payload->>'user_id')`

An index on `email` does not help `WHERE lower(email) = lower($1)`. The tree is sorted by the original string, including case. Looking up the lowercased form would require transforming every leaf entry at search time — which is exactly a sequential scan with a `Filter`, not an index seek.

So the structural limit is this: **the index keys and the query expression must be the same thing**. If the query transforms the column and the index does not, the planner falls back to reading candidates some other way and applying the expression row by row — `Filter: (lower(email) = ...)` after the row was already touched, instead of being the key that located the row.

An expression index fixes that at the root. It evaluates the expression once per write, stores the result as the leaf key, and lets matching queries seek that derived value directly.

A Concrete Example

Login and lookup paths almost always want case-insensitive email:

```
SELECT * FROM users
WHERE lower(email) = lower('Ada@example.com');
```

With only `CREATE INDEX ON users (email)`, the planner cannot seek with `lower(email)`. Typical plan shape:

```
Seq Scan on users
  Filter: (lower(email) = 'ada@example.com')
```

Or, if something else narrows rows first, an Index Scan still ends with the same expression as a post-filter. Either way the derived value is computed at read time across candidates.

With the expression index:

```
Index Scan using users_lower_idx on users
  Index Cond: (lower(email) = 'ada@example.com')
```

The lowercased form is already the key. One seek — no row-by-row lower() over the table.

Shapes that keep showing up:

Query needs	Index expression
Case-insensitive text	((lower(col))) or ((upper(col)))
Whole calendar day	((created_at::date))
JSON field lookup	((payload->>'id'))
Truncated timestamp	((date_trunc('day', created_at)))

Matching Is Exact

The planner uses an expression index only when it can see that the query predicate uses **the same expression** — same function, same cast, same argument order.

- `WHERE lower(email) = ...` — can use `((lower(email)))`
- `WHERE email = ...` — does **not** use that index; different key
- `WHERE lower(email) = lower($1)` — the index side matches; the right-hand `lower($1)` is a runtime constant once bound

Trap

Small rewrites break the match. `lower(email)` and `upper(email)` are different indexes. `(created_at::date)` and `date_trunc('day', created_at)` are not automatically interchangeable. Confirm with `EXPLAIN` after any change to the `WHERE` form.

Immutability

Postgres only allows expressions marked **IMMUTABLE** in an index definition. The indexed value must be a pure function of the row: same inputs → same key, forever, with no dependence on time, session settings, or other tables.

`lower(text)` is immutable. `now()`, `current_date`, and anything that reads another table are not. If the function could change result without the row changing, the index would silently go stale.

That rule is why “index `WHERE created_at > now() - interval '7 days'`” as an expression does not work as a rolling window — `now()` is not immutable. A **partial** index with a constant predicate, or a plain index on `created_at` plus a range query, is the right tool for time windows. Expression indexes compute a stable derived key; they do not encode a moving filter.

Expression + Partial, Expression + Composite

These cuts combine.

```
-- derived key, only for live rows
CREATE INDEX ON users ((lower(email)))
  WHERE deleted_at IS NULL;

-- tenant lead + derived second column
CREATE INDEX ON events (tenant_id, ((lower(event_type))));
```

Partial answers *which rows*. Composite answers *how many columns*. Expression answers *what value*. The same design tests from those topics still apply: leftmost prefix, implication proof for the partial predicate, equality before range.

2.10.3 Hands-On Inspection

Trade-offs

Consideration	What it means
Write cost	Expression evaluated on every insert/update that touches the inputs
Exact match	Query must use the same expression, or the index is ignored
Immutability	No volatile or stable-only functions as keys
Narrow reuse	An index on lower(email) does not help bare email = lookups
Stats	Planner stats are on the expression’s distribution — usually fine, but opaque vs a plain column

When to Reach for One

Use an expression index when the hot path **always** filters or joins on a derived form of a column — lowercased text, a cast, a JSON field — and that expression is immutable.

Skip it when you can store the derived value as a real column (generated columns are the honest alternative when many queries need the same derivation), when the function is not immutable, or when only occasional ad-hoc queries wrap the column.

2.10.4 Summary

At a glance

Structural gap.

An index on the raw column cannot seek a transformed value. The expression runs as a `Filter` instead.

Expression as key.

Evaluate once on write; store the derived leaf key.

Exact match.

The query expression must match the index expression, or the planner ignores it.

IMMUTABLE only.

The key must be a pure function of the row — no `now()`, no other tables.

Combines cleanly.

Works with partial membership and composite key order.

Design test.

Hot path always queries a stable derivation, not the raw column.

2.11 Chapter Summary

Ten topics, one contract: an index maps a key to `ctid`. Everything in this chapter is a different answer to what that key is, how it is organized, which rows participate, and when a read can stop at the leaf instead of visiting the heap.

Index = key → `ctid`

Which AM?

scalar equality/range → B-tree
 equality only → Hash
 huge append-ordered range → BRIN (page-range summaries)
 many discrete keys/row → GIN (entry tree + posting lists)
 continuous / overlap / KNN → GiST (predicate summaries)

Can the leaf answer alone? → covering / INCLUDE → index-only
 (heap skipped when VM says all-visible)

Which rows in the tree? → partial (WHERE on CREATE INDEX)
 How many columns in key? → composite (leftmost prefix)
 What value is the key? → expression (IMMUTABLE derivation)

Most important

2.1: B-tree.

The default sorted map. Keys live in leaves; internal pages are separators. Splits, HOT updates, and key order decide write cost and whether new values stay local in the leaf chain.

2.2: Index-only scans and covering.

When every needed column is in the index (keys or INCLUDE) and the visibility map marks the heap page all-visible, the read can stop at the leaf. Payload that is only selected belongs in INCLUDE, not in a widened key.

2.3: Hash.

Equality only: hash to a bucket, follow overflow chains. No ordering, no range. Wins when the predicate is pure equality and you do not need the sorted properties of B-tree.

2.4: BRIN.

Tiny summaries per block range. Cheap when physical order correlates with the key (append-mostly time or ID). A wrong correlation means the summary cannot exclude ranges — and you pay heap reads anyway.

2.5: GIN.

Inverted lists for many keys per row (arrays, jsonb, full text, trigrams). Entry tree plus posting lists; writes scatter; pending-list `fastupdate` trades lookup freshness for faster bulk insert.

2.6: GiST.

A tree of predicates for values that do not live on one sorted line — ranges, geometry, exclusion constraints, KNN. Summaries, not separators; multi-branch descent and recheck are normal, not failures.

2.7: Choosing an index.

Ask the column three questions: scalar or not (by access pattern); if not, discrete or continuous; what operation. GIN and GiST are hosts; operator classes plug in the type-specific answers (`pg_trgm` is `opclasses`, not a new engine).

2.8: Partial.

Membership filter on an existing AM — cost (size, cold writes) and conditional uniqueness, not a selectivity hack. The query `WHERE` must imply the index predicate; parameterized hot paths often break that proof.

2.9: Composite.

One concatenated key. Two single-column indexes force a runtime `Filter` or `BitmapAnd`; the composite encodes the intersection at build time. Leftmost prefix and equality-before-range decide reuse.

2.10: Expression.

Index the derived value the query actually asks for (`lower(email)`, a cast, a JSON field). Exact expression match; `IMMUTABLE` only. Combines with partial and composite cuts.

The whole chapter in one sentence.

Pick the access method from the shape of the data and the operation; then decide membership, key width, and whether the stored key is raw or de-

rived — so the planner seeks what the query means, instead of filtering what the index almost meant.

Three cuts on the same tree

After the AM is chosen, most design mistakes are about mixing up three independent questions:

Cut	Question	Failure mode if ignored
Partial	Which rows get an entry?	Huge cold majority in the tree, or uniqueness you cannot express
Composite	How many columns share the key?	Runtime <code>Filter</code> / <code>BitmapAnd</code> , or a tree that cannot serve the hot prefix
Expression	What value is stored as the key?	<code>Filter</code> over a raw-column index, or a volatile expression that cannot be indexed

Key point

These cuts stack. A partial composite expression index is one tree with three design decisions — not three features fighting each other. Get the AM right first; then answer membership, width, and derivation for the hot path you actually run.

MVCC and Concurrency

The previous chapter treated an index as a map from key to ctid, and left visibility in the heap. This chapter opens that heap tuple header. By the end you can take any tuple on any page and answer one question precisely: is this version visible to me, right now? That question is the machinery of MVCC — why versions instead of locks, what every header field means, what a snapshot is, how scans apply the visibility test, and how UPDATE and DELETE are implemented as versioning (with HOT as the MVCC optimization Chapter 2 already named).

3.1 Why Versions, Not Locks

Read consistency needs a single logical moment of the database while writers keep changing it. Lock-based designs make readers and writers take turns on one shared copy. MVCC removes that conflict by giving the reader an old version and the writer a new one.

3.1.1 What It Is

The Problem: a Consistent View of a Changing Database

Consider a report that sums every account balance while the application keeps transferring money between accounts.

```
-- Session A (report)
SELECT sum(balance) FROM accounts;
      balance - 50 WHERE id = 1;

      balance + 50 WHERE id = 2;

-- Session B (transfers)
UPDATE accounts SET balance =
      balance + 50 WHERE id = 1;
UPDATE accounts SET balance =
      balance - 50 WHERE id = 2;
COMMIT;
```

A transfer moves money; it never creates or destroys it, so the true total never changes. But if the report reads account 1 *before* the transfer and account 2 *after* it, it will count the 50 twice. A correct reader needs to see the database **as of a single logical moment**, even though that “moment” lasts as long as the scan — milliseconds or hours.

That is **read consistency**. The question is how to provide it.

The Lock-Based Answer: Two-Phase Locking

The classical answer is **two-phase locking** (2PL). The name is literal: a transaction’s life has two phases, and it never mixes them.

Growing phase. Every time the transaction touches data, it *acquires* a lock and keeps it:

- A **shared (S)** lock to read — “I need this row stable.”
- An **exclusive (X)** lock to write — “I need to change this row.”

Locks only accumulate. You never release one during growing.

Shrinking phase. Locks are released. In **strict 2PL** — the form that actually gives you serializable-looking reads — shrinking happens **all at once**, at commit or abort. Nothing is released early. A long report that read a million rows holds a million shared locks until it finishes.

That protocol is correct. A report holding shared locks on every row it touched cannot see a half-finished transfer: the transfer cannot get its exclusive locks until the report releases, or the report waits for the transfer. One of them goes first; consistency is preserved.

Where the Trouble Starts: Same Data, Different Intentions

Correctness is not the problem. The trouble starts when two transactions touch the **same data in different modes**:

- A reader wants it stable → **S** lock.
- A writer wants to change it → **X** lock.
- **S and X conflict**, so one must wait for the other.

Because locks are held until the end of the transaction (strict shrinking), that wait can be long. A report that runs for minutes parks every writer that touches any of those rows behind it.

It can also form a cycle — a **deadlock** — even when nobody “did anything wrong”:

```
T1 holds lock on row A, wants row B
T2 holds lock on row B, wants row A
→ deadlock (the database must abort one)
```

So the same data, accessed with different intentions, forces transactions to **take turns**. That is not a tuning failure. It is the structure of 2PL when there is only one copy of each row: readers who need it stable and writers who need to change it cannot proceed at the same time.

The structural costs compound from there:

1. **Reader–writer conflict is built in.** Shared vs exclusive on the same copy.

2. **Lock footprint scales with read size.** A million rows → a million locks, or escalation to the whole table.
3. **New deadlock classes.** Cross-row wait cycles like the one above.
4. **Throughput collapses exactly when you need it.** OLTP writers plus analytical readers — the combination real systems always have — is what 2PL handles worst.

The MVCC Answer: Stop Sharing a Single Copy

That conflict — reader wants the row stable, writer wants to change it, only one copy exists — is exactly why **Multi-Version Concurrency Control** exists. MVCC removes the conflict by removing the sharing:

Key point

A write never overwrites a row. It creates a **new version** of it. Old versions stay around for as long as some transaction might still need them. Each transaction reads the version that was current as of *its own* point in time.

The reader gets an old, immutable version. The writer creates a new one. They no longer compete for the same copy, so they no longer need S vs X on that row data. A writer creating version $N+1$ cannot disturb a reader looking at version N .

Two writers on the **same** row still conflict — there is still only one “latest” version to own — and they still take turns. MVCC solved the *read–write* problem, not the *write–write* problem.

PostgreSQL’s rule of thumb:

Interaction	Blocks?
Reader vs. reader	No
Reader vs. writer	No — that is the point of versions
Writer vs. reader	No
Writer vs. writer, <i>same row</i>	Yes — the second waits for the first to commit or abort
Writer vs. writer, <i>different rows</i>	No

3.1.2 Why This Matters

Where Do the Old Versions Live?

Every MVCC system has to answer this, and the answer shapes the whole engine.

Design	Latest	Old versions	Examples
In-place + undo	In the table	Separate undo/rollback; readers reconstruct old versions	Oracle, InnoDB
Append-only heap	In the table	In the table itself , as ordinary tuples	PostgreSQL
Version store	In the table	Separate store (e.g. tempdb)	SQL Server (SI)

PostgreSQL’s choice has consequences that run through the rest of this chapter and into Vacuum later:

Benefits

- **Rollback is essentially free.** An aborted transaction’s tuples are simply never considered visible; nothing has to be “undone”. Aborting is a status bit, not a data operation.

- **No undo segment to size or run out of.** Readers never reconstruct anything; they just *skip* tuples they are not allowed to see.
- **Crash recovery is simple.** No undo phase for row data.

Costs

- **Bloat.** Dead versions occupy table (and index) space until they are removed.
- **A cleanup process is mandatory.** That is what `VACUUM` exists for.
- **Write amplification.** An `UPDATE` writes a whole new tuple and, unless `HOT` applies (Section 3.5), new entries in *every* index.
- **Readers pay a per-tuple check.** Every tuple a scan touches must pass a visibility test, even tuples that turn out to be dead.

What MVCC Does Not Do

- **It does not remove write–write conflicts.** Two transactions updating the same row still serialize on that row.
- **It does not remove table-level locks.** A plain `SELECT` takes an `ACCESS SHARE` lock on the table. That lock only conflicts with `ACCESS EXCLUSIVE`, which `ALTER TABLE`, `DROP TABLE`, `VACUUM FULL`, and others take. So DDL *can* block readers, and readers can block DDL.
- **It does not give you serializability by itself.** MVCC gives you *snapshot* semantics. Full serializability needs extra machinery (Serializable Snapshot Isolation) layered on top — a later topic in this chapter.

3.1.3 Summary

At a glance

Read consistency needs one logical moment.

A long scan must see the database as of a single point in time while writers keep changing it.

Strict 2PL grows, then shrinks once.

Growing acquires S on read and X on write; shrinking releases everything at commit or abort.

Trouble is same data, different intentions.

S vs X conflict forces waiting; wait cycles become deadlocks. Readers and writers take turns on one shared copy.

That conflict is why MVCC exists.

Readers get an old immutable version; writers create a new one. Read–write no longer competes for the same copy.

Write–write on the same row still blocks.

MVCC solved read–write, not two writers racing for the latest version.

PostgreSQL keeps old versions in the heap.

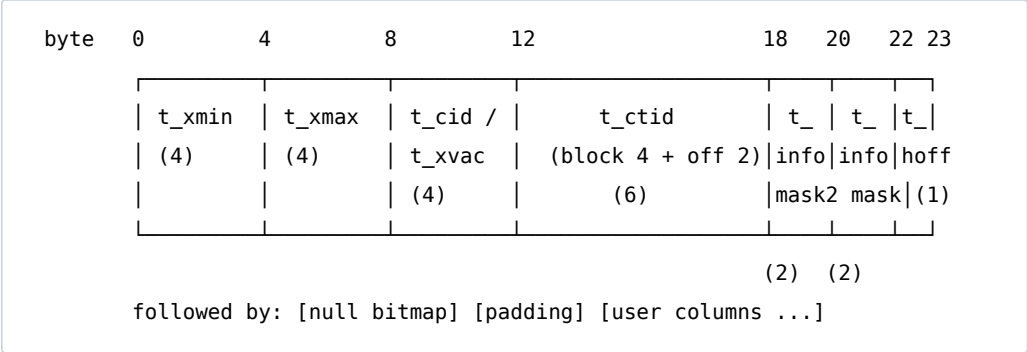
Free rollback, mandatory vacuum, write amplification, and a per-tuple visibility check on every scan. DDL locks and SSI are separate layers.

3.2 Tuple Headers

Every heap tuple starts with a fixed header. The fields that matter for visibility — `xmin`, `xmax`, command IDs, `ctid`, and hint bits — live here. Commit status itself lives in `pg_xact`; the header names the transactions and caches their outcomes.

3.2.1 What It Is

Every heap tuple starts with a fixed header, `HeapTupleHeaderData`. It is 23 bytes, padded to 24 on typical platforms before the null bitmap and user data begin.



xmin and xmax: Who Created It, Who Ended It

Field	Meaning
xmin	The transaction ID (XID) that inserted this tuple version.
xmax	The XID that deleted or updated this version (or, as we'll see, <i>locked</i> it). 0 means "no one has".

Those two numbers are almost the whole visibility story — but they only name *which* transactions. Whether each of those transactions **committed or aborted** is not stored in the tuple. That answer lives elsewhere: in `pg_xact`.

What Is pg_xact?

pg_xact (historically **clog**, the commit log) is PostgreSQL's small shared table of transaction outcomes. For every real XID it stores a status in **two bits**:

Status	Meaning
in progress	Transaction has not finished yet
committed	Transaction committed
aborted	Transaction rolled back (or crashed before commit)
sub-committed	Used for subtransactions (savepoints)

It lives under `PGDATA/pg_xact/` as a compact on-disk bitmap, kept in shared memory while the server runs. Every `COMMIT` and `ABORT` updates it. It is the **source of truth** for “did XID *N* succeed?” — the heap tuple never duplicates that fact until a **hint bit** caches it (below).

Informally, a version is alive from “xmin committed” until “xmax committed”. Both of those checks go through pg_xact (or a hint that already recorded its answer).

Trap

XIDs are assigned lazily. A transaction does not get an XID at `BEGIN`. Read-only transactions use a *virtual* transaction ID and never consume a real XID. A real 32-bit XID is assigned on the first write. That is why XID order is **not** commit order (Section 3.3).

```
SELECT xmin, xmax, ctid, * FROM accounts WHERE id = 1;
```

cmin and cmax: Ordering Statements Inside One Transaction

xmin / xmax answer *which transaction*. They cannot order **two statements of the same transaction**.

```
BEGIN;                                -- command IDs start at 0
INSERT INTO t VALUES (1);            -- cid 0: creates the row (cmin = 0)
SELECT * FROM t;                      -- cid 1: must SEE the insert
UPDATE t SET x = x + 1;                -- cid 2: must NOT re-read the row it just
    wrote
COMMIT;
```

Without a finer clock, the UPDATE would see its own new version, update it again, loop forever — the classic **Halloween problem**.

The fix: each statement gets a **command ID** (cid), a counter that ticks once per command inside the transaction.

Field	Means
cmin	Which command inserted this version
cmax	Which command deleted this version

Rule of thumb (own transaction only):

- See my insert only if cmin < current cid (an *earlier* command created it).
- Treat my delete as done only if cmax < current cid (an *earlier* command deleted it).

Other transactions never look at cmin / cmax. They only care whether *my whole transaction* committed. So these fields exist solely so **one transaction can see its own earlier statements, but not its own current write**.

Storage, short version: both share one 4-byte slot (t_cid). Usually only one is needed. If the *same* transaction both inserts and deletes a version, PostgreSQL stores a **combo CID** (an index into a private backend array) and sets HEAP_COMBOCID. You can ignore that detail until you read the source; the rule above is what matters.

ctid: Where Is the Next Version?

`t_ctid` is a **tuple identifier** (block number, line pointer offset).

- On the **newest** version of a row, `t_ctid` points to **itself**.
- On an **old** version that has been updated, `t_ctid` points to the **next (newer) version**.

So versions of a row form a singly linked list, from oldest to newest:

```
(0,1) —ctid→ (0,4) —ctid→ (2,7) —ctid→ (2,7)
oldest                                         newest (points to itself)
```

Two important notes:

- The chain is walked **old** → **new** only. Nothing points backwards. Readers arrive through an index or a scan and need to find the right version (Section 3.4).
- The user-visible `ctid` system column is **not a stable row identifier**. It changes on every non-HOT update and after `VACUUM FULL`. Never store it as a key.

3.2.2 Why This Matters

Hint Bits: Caching What `pg_xact` Already Knows

Visibility still needs “did this XID commit?” for every `xmin` / `xmax` it sees. The authoritative answer is in `pg_xact`.

Without a cache, every scan of every tuple looks up `xmin` and `xmax` in `pg_xact`. On a large table that is a flood of extra reads against the same shared structure — correct, but expensive.

Hint bits are flags in the tuple header (`t_infomask`) that **cache** the `pg_xact` answer next to the row:

Flag	Cached answer
HEAP_XMIN_COMMITTED	<code>xmin</code> committed
HEAP_XMIN_INVALID	<code>xmin</code> aborted (or crashed)
HEAP_XMAX_COMMITTED	<code>xmax</code> committed
HEAP_XMAX_INVALID	<code>xmax</code> unset, or that transaction aborted

How they work:

1. First reader after a commit finds no hint → looks up `pg_xact` → **writes the answer into the tuple.**
2. Every later reader sees the hint and **skips `pg_xact` entirely.**

So: `pg_xact` remains authoritative; hint bits are a **per-tuple cache** of that authority. That is the whole design.

Without hint bits	With hint bits
Every visibility check hits <code>pg_xact</code>	First check pays; the rest are local to the page
Commit-log traffic scales with <i>reads</i>	Commit-log traffic stays tied to <i>commits/aborts</i>
Hot <code>pg_xact</code> pages become a bottle-neck	Heap pages carry their own cached answers

Key point

Hint bits turn a repeated shared lookup into a one-time cost per tuple version. `pg_xact` stays the source of truth; the tuple carries a cache of its answer.

Side effects worth knowing:

- 1. A `SELECT` can dirty pages. The first read after a bulk load sets many hint bits → buffers become dirty → writes. That is the “my read-only query caused I/O” surprise.
- 2. Usually not WAL-logged. They only cache what `pg_xact` already recorded. Exception: data checksums or `wal_log_hints` — then the first hint-bit change after a checkpoint may log a full-page image.
- 3. Set only when the answer is durable. A “committed” hint must wait until the commit record is flushed. Otherwise a crash could leave a page claiming commit when recovery decides abort.

Other Flags (Quick Reference)

Still in `t_infomask` / `t_infomask2`, but not the hint-bit story:

Flag	Meaning
HEAP_XMAX_IS_MULTI	xmax is a MultiXact (several lockers/updaters)
HEAP_XMAX_LOCK_ONLY	xmax is a row lock , not a delete/update
HEAP_COMBOCID	t_cid holds a combo command ID
HEAP_UPDATED	This version is the result of an UPDATE
HEAP_HOT_UPDATED	Successor is a heap-only tuple
HEAP_ONLY_TUPLE	No index entry points here (Section 3.5)
HEAP_KEYS_UPDATED	Update changed key columns (FK locking)

Together, `HEAP_XMIN_COMMITTED` and `HEAP_XMIN_INVALID` mean **frozen** (`HEAP_XMIN_FROZEN`): visible to everyone; visibility no longer depends on the `XID`. That is how `VACUUM` survives 32-bit `XID` wraparound (deep dive later). Older versions overwrote `xmin` with `FrozenTransactionId`; modern versions keep the original `xmin` and flag it.

3.2.3 Hands-On Inspection

```
CREATE EXTENSION IF NOT EXISTS pageinspect;

CREATE TABLE acct (id int PRIMARY KEY, bal int);
INSERT INTO acct VALUES (1, 100);

SELECT lp, t_xmin, t_xmax, t_field3 AS t_cid, t_ctid,
       (heap_tuple_infomask_flags(t_infomask, t_infomask2)).raw_flags AS
       flags
FROM   heap_page_items(get_raw_page('acct', 0));
```

Illustrative output:

lp	t_xmin	t_xmax	t_cid	t_ctid	flags
1	750	0	0	(0,1)	{HEAP_XMAX_INVALID}

The tuple is alive, `t_ctid` points to itself, and no commit hint is set yet, because no one has examined it since the inserting transaction committed. After a `SELECT * FROM acct;` the same query would typically also show `HEAP_XMIN_COMMITTED`.

3.2.4 Summary

At a glance

The header names transactions; `pg_xact` judges them.

`xmin/xmax` say who created and who ended a version. `pg_xact (clog)` under `PGDATA/pg_xact/` stores two bits per `XID` — the source of truth for commit or abort.

`cmin/cmax` order statements inside one transaction.

Command IDs stop a transaction from re-reading its own current write (Halloween problem). Other transactions ignore them.

`ctid` links old to new.

Newest points at itself. The user-visible `ctid` is not a stable key.

Hint bits cache `pg_xact` on the tuple.

First reader pays and may dirty the page; later readers skip the commit log. That is the scalability problem they solve.

A SELECT can cause writes.

Setting hint bits dirties buffers. With checksums or `wal_log_hints`, the first change after a checkpoint may also WAL-log a full-page image.

3.3 Snapshots

A snapshot is not a copy of the data. It is a small note that says which transactions were still running at a moment in time — and that note decides what every visibility check will accept.

3.3.1 What It Is

The One-Sentence Idea

A **snapshot** is a small note a transaction takes that says: **at this moment, these transactions were still running. Treat everything else as finished or not started yet.**

It is **not a copy of the data**. It is just three things.

Analogy: imagine a whiteboard listing who is currently in a meeting. You take a photo of the whiteboard. Later you use the photo to decide whose work is “done” (and you can trust it) and whose is “still in progress” (and you ignore it). The photo does not change, even if the whiteboard does.

Why We Need It

Every row version has a header with `xmin` (creator) and `xmax` (deleter). To decide whether you can see a version, you ask: had the transaction that created it already finished when my snapshot was taken? The snapshot is what answers that question.

The Three Parts of a Snapshot

Part	Meaning
xmin	The oldest transaction still running. Everything below it is finished .
xmax	The next transaction ID that has not been handed out yet. Everything at or above it started after the snapshot .
xip[]	The list of transactions running right now between xmin and xmax.

Trap

Do not mix them up. The snapshot's xmin/xmax are **different** from the row's xmin/xmax. They share names but mean different things. Row fields name who created or deleted this version; snapshot fields bound the range of XIDs the reader cares about.

```
SELECT pg_current_snapshot();  -- example result: 100:104:100,102
```

The format is xmin:xmax:xip_list.

Reading 100:104:100,102

Transaction ID	Status in snapshot	See its changes?
below 100	Finished	Check commit or abort
100	In xip → running	No
101	Not in list → finished	Check commit or abort
102	In xip → running	No
103	Not in list → finished	Check commit or abort
104 and above	Not started yet	No

A transaction counts as **still running** (so you ignore its changes) if:

1. its ID is $\geq x_{\max}$, or
2. its ID is in xip[].

Anything else is **finished**. For finished transactions, PostgreSQL then checks whether they committed (visible) or aborted (invisible), via a hint bit or pg_xact.

Why the xip List Exists

Why not just say “I see everything with an ID lower than mine”? Because **ID order is not the same as commit order**:

```
T100 starts and writes    (gets ID 100)
T101 starts and writes    (gets ID 101)
T101 commits
--- snapshot taken here ---    → xmin=100, xmax=102, xip={100}
T100 commits (later)
```

The reader with this snapshot must:

- **see T101** (it committed before the snapshot)
- **not see T100** (it was still running), even though $100 < 101$

One simple threshold cannot express “101 yes, 100 no”. The `xip` list can.

3.3.2 Why This Matters

When Is the Snapshot Taken?

This is what decides how your isolation level behaves.

Isolation level	When taken	Result
READ COMMITTED (default)	At the start of every statement	Each statement sees the latest committed data. Two queries in one transaction can see different data.
REPEATABLE READ	Once , at the transaction's first query	The whole transaction sees one frozen view.
SERIALIZABLE	Once, same as repeatable read	Same frozen view, plus extra tracking to detect conflicts (SSI).

Notes:

- PostgreSQL treats `READ UNCOMMITTED` as `READ COMMITTED`. Dirty reads never happen.

- The snapshot is taken at the **first real query**, **not** at BEGIN. If you run BEGIN and then wait, there is no snapshot yet.

Example: the Same Steps Under Two Levels

Step	Session A	Session B
1	BEGIN;	
2	SELECT bal FROM acct WHERE id=1; → 100	
3		UPDATE acct SET bal=90 WHERE id=1; COMMIT;
4	SELECT bal FROM acct WHERE id=1; → ?	

- READ COMMITTED → step 4 returns 90. It takes a **new snapshot**, and B has already committed.
- REPEATABLE READ → step 4 returns 100. It **reuses the snapshot from step 2**, where B was still running.

What if A tries to UPDATE that row?

Level	What happens
READ COMMITTED	Waits for B, then re-checks the WHERE against the newest version.
REPEATABLE READ	Fails with could not serialize access due to concurrent update. A's snapshot says the balance is 100, but the real latest version is B's.

Small Detail About Functions

- A `VOLATILE` function gets a fresh snapshot for each statement inside it (under read committed).
- `STABLE` and `IMMUTABLE` functions use the snapshot of the statement that called them.
- A cursor keeps the snapshot it had when it was declared.

The Cleanup Horizon

Each backend publishes the `xmin` of its snapshot in shared memory, and `VACUUM` takes the **lowest** of those values as the **cleanup horizon**.

A snapshot's `xmin` means “I might still need to see anything that was alive at that point.” Every running transaction is making that promise, so PostgreSQL cannot remove an old row version if *any* transaction might still need it. The oldest promise decides what is safe to delete.

Example. Three transactions are running:

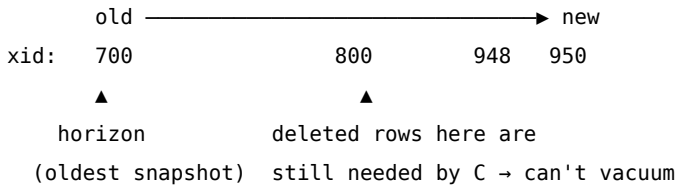
Transaction	Snapshot xmin
A (short query)	950
B (short query)	948
C (report running for 3 hours)	700

The horizon is the lowest value, **700**.

Now take a dead row version: it was deleted by transaction 800, and 800 committed.

- Transactions A and B started later, so they cannot see it.
- Transaction C's snapshot is older than 800, so **C can still see it**.

`VACUUM` therefore cannot remove it. Everything deleted after 700 is stuck the same way, in every table, until C finishes.



When C commits, the lowest `xmin` jumps to 948, the horizon moves forward, and `VACUUM` can reclaim everything older than 948.

Key point

One long-running transaction, or one session left idle in transaction, holds the horizon back for the **whole cluster**. Dead rows pile up in every table, even ones that transaction never touched. That is bloat, and it is one of the most common causes of it in production.

3.3.3 Monitoring

How to find the culprit:

```
SELECT pid, state, xact_start, backend_xmin
FROM pg_stat_activity
WHERE backend_xmin IS NOT NULL
ORDER BY age(backend_xmin) DESC;
```

The row at the top, with the oldest `backend_xmin`, is the one holding the horizon back. Replication slots and prepared transactions can also hold the horizon back, so check those too.

Other Snapshot Types (Optional)

Normal queries use the MVCC snapshot described above. The code also has a few special ones:

Type	What it does
SnapshotDirty	Also sees uncommitted rows. Used by unique-index checks to detect in-flight conflicting inserts.
SnapshotSelf	Sees the current transaction's own changes.
SnapshotAny	Sees every tuple regardless of status.
VACUUM's check	Uses no snapshot. It compares against the cleanup horizon.

3.3.4 Summary

At a glance

A snapshot is `xmin:xmax:xip[]`.

It says who counts as finished from the reader's point of view. It is not a copy of data.

Running means $\geq xmax$ or in `xip[]`.

Everything else is finished; then check commit or abort via hint bits or `pg_xact`.

`xip` exists because ID order $\neq$ commit order.

A lower XID can still be running after a higher one has committed.

READ COMMITTED snapshots per statement.

REPEATABLE READ and SERIALIZABLE take one snapshot per transaction, at the first query — not at BEGIN.

The lowest published snapshot xmin is the cleanup horizon.

One long transaction or idle in transaction session can block VACUUM cluster-wide. Find it with `pg_stat_activity.backend_xmin`.

3.4 Visibility Rules

Visibility is two questions: has the insert already happened for me, and has the delete already happened for me? Indexes carry no visibility information — every scan reaches the heap (or the visibility map) and each transaction builds its own visible set from its snapshot.

3.4.1 What It Is

The Helper Question: Is This XID in My Snapshot?

Everything reduces to one small predicate:

```
XidInMVCCSnapshot(xid, snapshot):  
    if xid < snapshot.xmin:    return false    -- finished before snapshot  
    if xid >= snapshot.xmax:   return true     -- started after snapshot  
    return xid ∈ snapshot.xip      -- in-progress at snapshot time
```

“False” does **not** mean “committed”. It means “finished”, and we still need `pg_xact` (or a hint bit) to learn *how* it finished.

The Decision Procedure

This is a simplified rendering of the function `HeapTupleSatisfiesMVCC` in `heapam_visibility.c` (under `src/backend/access/heap/`). We ignore subtransactions, MultiXacts, and legacy `HEAP_MOVED_*` handling.

```
visible(tuple, snapshot):

    -- Step 1: was the tuple created, as far as I'm concerned? --
    if xmin is MY transaction:
        if tuple.cmin >= snapshot.curcid: return false
        # else: created by an earlier command of mine → go to step 2
    else:
        if HEAP_XMIN_INVALID:           return false
        if XidInMVCCSnapshot(xmin):     return false
        if not HEAP_XMIN_COMMITTED:
            if pg_xact says xmin committed: set HEAP_XMIN_COMMITTED
            else: set HEAP_XMIN_INVALID; return false

    -- Step 2: has it been deleted, as far as I'm concerned? --
    if xmax is invalid or LOCK_ONLY:    return true
    if xmax is MY transaction:
        return (tuple.cmax >= snapshot.curcid)
    if XidInMVCCSnapshot(xmax):         return true
    if xmax not committed (hint / pg_xact):
        set HEAP_XMAX_INVALID;          return true
    set HEAP_XMAX_COMMITTED;            return false
```

Read it as two questions:

1. **Had the insert already “happened” for me?** (If not, the tuple is from the future or from an aborted transaction.)
2. **Had the delete already “happened” for me?** (If so, the tuple is from the past.)

A tuple is visible iff the answer to (1) is yes and the answer to (2) is no.

The Truth Table (Other Transactions' Tuples)

For a tuple *not* created by my own transaction:

xmin status	xmax status	Visible?	Why
Aborted	—	No	The insert never happened
In progress / after snapshot	—	No	The insert hasn't happened <i>yet</i> , for me
Committed before snapshot	None	Yes	Normal live row
Committed before snapshot	Lock only	Yes	Locking is not deleting
Committed before snapshot	Aborted	Yes	Delete was rolled back
Committed before snapshot	In progress / after snapshot	Yes	Delete hasn't happened yet, for me
Committed before snapshot	Committed before snapshot	No	Deleted (or replaced by an update)

The last row is the only one where a committed tuple becomes invisible, and it is also the answer to “where did the old version of my updated row go?”.

A Worked Example

Snapshot: 100:104:100,102 ($x_{\min}=100$, $x_{\max}=104$, $x_{ip}=\{100,102\}$). Assume XIDs 101 and 103 committed, and 99 committed long ago.

Tuple	$x_{\min}$	$x_{\max}$	Verdict	Reasoning
A	99	0	Visible	Created by a long-finished committed transaction, never deleted
B	101	0	Visible	101 finished and committed; not in x_{ip}
C	100	0	Invisible	100 is in x_{ip} (still running when I looked)
D	99	102	Visible	Deleter 102 is in x_{ip} , so for me the delete has not happened
E	99	101	Invisible	Deleter 101 committed before my snapshot
F	105	0	Invisible	$105 \geq x_{\max}$; started after my snapshot
G	101	103	Invisible	Both finished and committed: a version already superseded

Rows from My Own Transaction: $c_{\min}/c_{\max}$

When the tuple's $x_{\min}$ (or $x_{\max}$) is *my* transaction, snapshots are useless, because my own changes are never “in progress” from my view. Instead we compare command IDs against the snapshot's *curcid* (the ID of the command currently running):

- A tuple I inserted is visible only to commands **after** the one that inserted it ($c_{\min} < \text{curcid}$).
- A tuple I deleted is **still visible** to the command doing the deleting, and any earlier one, and gone for later commands ($c_{\max} < \text{curcid}$).

This is what stops `UPDATE t SET x = x + 1` from re-reading the new versions it just created (the Halloween problem), while still letting the *next* statement see them.

3.4.2 Why This Matters

The One Idea

To decide visibility you need the tuple's `xmin` and `xmax`. Those live **only in the heap tuple header**. An index entry holds just two things:

```
index entry = ( key value , TID )           e.g. ( id=1 , (0,3) )
               └─ where the row is in the heap: page 0, slot 3
```

So an index can tell you “a row with `id=1` exists at `(0,3)`”, but it cannot tell you whether **you** are allowed to see it. To find out, you have to go to the heap and read the header. Every scan type is a different way of reaching the heap tuples and then running the visibility test.

Key point

Same page, different answers. Two transactions can read the same heap page at the same time and disagree about which tuples are visible. Each one runs the test against **its own snapshot**. The page does not store “the” visible set — each transaction builds its own.

This is the same contract Chapter 2 started from: the index maps a key to a `ctid`; visibility lives in the heap (Section 2.2).

Example: Why the Index Cannot Decide

acct has an index on id. Row id=1 was updated, so there are two versions in the heap, and (without HOT) two index entries:

```
Index:  ( id=1 , (0,1) )    ( id=1 , (0,2) )
Heap:   (0,1) old: bal=100, xmax=751          (0,2) new: bal=90, xmin=751
```

Suppose transaction 751 committed *after* your snapshot. Both index entries match WHERE id = 1, and the index cannot choose between them. Only the heap check can:

- (0,1): its deleter 751 hasn't happened yet from your view, so **visible**.
- (0,2): its creator 751 hasn't happened yet from your view, so **invisible**.

How Each Scan Does It

Scan	What it does for visibility
Sequential	Read every page; for each tuple, run the test. Page-at-a-time: lock once, test all tuples on the page under this snapshot, remember which are visible, unlock.
Index	Index gives a TID → fetch that heap tuple → run the test. Fail → discard, ask the index for the next TID.
HOT chain	Index points at the root of a version chain (Section 3.5). Walk <code>ctid</code> links; return the first version visible to your snapshot .
Kill bit (<code>LP_DEAD</code>)	If a heap tuple is dead to <i>everyone</i> (not just you), mark the index entry dead so later scans skip the heap visit. Free cleanup along the way.
Bitmap heap	Collect matching TIDs into a bitmap, then visit those heap pages in physical order and test each tuple — like a seq scan limited to useful pages.
Index-only	Want to skip the heap, but visibility lives there. Use the visibility map : bit set = every tuple on the page is visible to everyone → skip heap; bit clear → heap fetch + normal test. <code>VACUUM</code> sets those bits (Section 2.2).

What to keep:

1. Indexes never decide visibility — only the heap (or the VM shortcut) does.
2. Each transaction computes **its own** visible set on a page from its snapshot.
3. HOT / kill bits / VM are optimizations on top of the same test, not different rules.

Writes: Same Rules, Then Waiting

UPDATE and DELETE find rows with the **same** visibility rules, then hit a new problem: the row you want to change may be changed by someone else right now.

What the other writer did	What you do
Still running	Wait for it to finish
Aborted	Its change never happened → go ahead
Committed, and you are in READ COMMITTED	Follow the <code>ctid</code> chain to the newest version and re-check your WHERE against it. If it still matches, update that version
Committed, and you are in REPEATABLE READ or higher	Your snapshot is fixed and cannot see the new version → fail with <code>could not serialize access due to concurrent update</code>

Key point

This is the point where MVCC stops being “readers never wait” and writers meet each other. Visibility picks the version you are allowed to see; **isolation level** decides what happens when that version is no longer the latest.

3.4.3 Summary

At a glance

`XidInMVCCSnapshot` answers “**still running from my view?**”
Finished $\neq$ committed; check `pg_xact` / hint bits next.

Visibility is insert happened and delete has not.
Two questions, both answered relative to your snapshot (or `cmin/cmax` for your own transaction).

Own-transaction rows use command IDs.

`cmin/cmax` vs `curcid` stop the Halloween problem.

Indexes have no visibility.

Every scan reaches the heap or the visibility map. Each transaction builds its own visible set from its snapshot — the same contract as [Chapter 2](#).

Writes wait, then re-check or abort.

If the other writer is still running, wait. On commit, read committed re-checks the newest version; repeatable read and higher raise a serialization failure.

3.5 UPDATE and DELETE as Versioning

There is no in-place update in PostgreSQL. DELETE stamps `xmax`; UPDATE marks the old version and inserts a new one. [Chapter 2](#) introduced HOT as a page-layout trick; with MVCC in hand we can see the problem it actually solves.

3.5.1 What It Is

DELETE: Stamp It, Don't Remove It

A DELETE does not remove anything. It:

1. Finds the visible tuple version.
2. Writes the deleting transaction's XID into `xmax` (and the command ID into `cmax`).
3. Leaves `t_ctid` pointing at itself, since there is no newer version.
4. Logs the change to WAL.

The tuple stays physically in place. It becomes invisible to any snapshot that treats the deleter as committed, and remains visible to everyone else. Only when *no* snapshot could still see it does it become **dead**, eligible for removal by `VACUUM` or by page pruning.

If the deleting transaction **aborts**, the `xmax` value stays in the header but is ignored: the status says "aborted", so the tuple is treated as never deleted (and the next reader to notice sets `HEAP_XMAX_INVALID` as a cache). There is nothing to roll back on the page.

UPDATE: a Delete Plus an Insert

An UPDATE is, physically, exactly this:

1. **Mark the old version as deleted:** set its `xmax` = my XID.
2. **Insert a new version** with the new column values, `xmin` = my XID, flagged `HEAP_UPDATED`.
3. **Link them:** set the old version's `t_ctid` to point at the new version.
4. **Insert index entries** for the new version in each index that needs one (see below).

Using the `acct` example, `UPDATE acct SET bal = 90 WHERE id = 1;` run by transaction 751:

BEFORE	AFTER (transaction 751, before commit)
lp1: xmin=750 xmax=0 ctid=(0,1) bal=100	lp1: xmin=750 xmax=751 ctid=(0,2) ← old lp2: xmin=751 xmax=0 ctid=(0,2) ← new bal=90 (self)

Both rows exist on the page simultaneously. Which one a reader sees depends entirely on its snapshot and on 751's fate:

Reader's snapshot says 751 is...	Sees
in progress, or started after the snapshot	lp1 (bal = 100)
committed before the snapshot	lp2 (bal = 90)
aborted	lp1: lp2's xmin is aborted so lp2 is never visible, and lp1's delete is ignored

Illustrative pageinspect output after the commit and a later read:

lp	t_xmin	t_xmax	t_ctid	flags
1	750	751	(0,2)	{HEAP_XMIN_COMMITTED,HEAP_XMAX_COMMITTED,HEAP_HOT_UPDATED}
2	751	0	(0,2)	{HEAP_XMAX_INVALID,HEAP_UPDATED,HEAP_ONLY_TUPLE}

Key point

There is no in-place update in PostgreSQL. Even UPDATE t SET flag = flag on one boolean writes a complete new row version.

What Happens to the Indexes

Index entries point at a **heap location** (TID). A new row version lives at a **new** TID, so, without HOT, *every* index on the table needs a new entry pointing to it. The old entries remain, pointing at the old version, until `VACUUM` removes them.

That is the real cost of “update = delete + insert”:

- A table with 8 indexes gets **8 new index entries per updated row**, including indexes on columns that did not change.
- Index bloat grows alongside heap bloat.
- Each new entry is more WAL.

This is the motivation for HOT — the same optimisation Section 2.1 named from the index side.

`xmax` Is Not Only for Deletes: Row Locks

`SELECT ...FOR UPDATE`, `FOR NO KEY UPDATE`, `FOR SHARE`, and `FOR KEY SHARE` also write into `xmax`, marking it with `HEAP_XMAX_LOCK_ONLY`. A row lock is stored **in the row**, not in a shared lock table. That is why PostgreSQL can lock millions of rows without a lock-table explosion, and also why locking a row **dirtyes its page**.

When several transactions hold compatible locks on the same row (or a locker coexists with an updater), a single XID won't fit in `xmax`. PostgreSQL then stores a **MultiXact ID**, an ID that maps to a *set* of transactions, and sets `HEAP_XMAX_IS_MULTI`.

This is why the visibility algorithm checks `LOCK_ONLY` before treating `xmax` as a delete.

3.5.2 Why This Matters

HOT Revisited: an MVCC Optimization

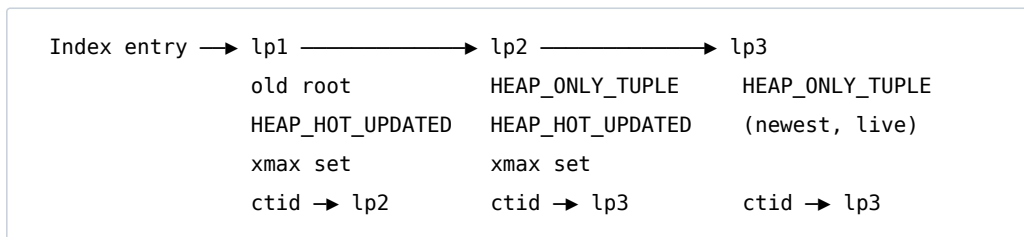
Chapter 2 introduced **Heap-Only Tuples** as a page-layout idea. With MVCC in hand we can see what problem they actually solve.

Most updates change columns that no index covers. For those, creating new index entries is pure waste: the index key hasn't changed, and the only reason the entry would need to change is that the row moved.

The trick: if the new version can be placed **on the same heap page** as the old one, the index can keep pointing at the *old* tuple's line pointer, and the heap itself forwards the reader to the newest version.

Conditions for a HOT update

1. **No indexed column changed.** More exactly, no column used by *any* index on the table, including columns inside expression indexes and partial-index predicates, has a changed value. (Since PostgreSQL 16, changes to columns used only by *summarizing* indexes such as BRIN no longer block HOT.)
2. **The new version fits on the same page.** If the page is full, the update falls back to a regular one.



- The **index entry still points to lp1**, the root of the chain.
- lp2 and lp3 are **heap-only**: no index has an entry for them. They are marked `HEAP_ONLY_TUPLE`.
- Each superseded version is marked `HEAP_HOT_UPDATED`, and the `ctid` links form the chain.
- An index scan reaches lp1 and walks the chain, applying the visibility test at each member, until it finds one visible to its snapshot (Section 3.4).

Why This Is an MVCC Optimization

The old versions in the chain are exactly the dead versions MVCC creates. Because they have no index entries, they can be removed **without touching any index and without a full VACUUM**. PostgreSQL does this opportunistically, in **page pruning**: when a backend reads a page that is nearly full and contains prunable tuples, it can clean the page on the spot:

- Dead heap-only tuples are removed and their line pointers freed.
- The chain's root, which an index still points at, cannot be freed, so it is converted to an `LP_REDIRECT` line pointer that forwards directly to the first live version. The root's data space is reclaimed; only the 4-byte pointer remains.
- If the whole chain is dead, the root becomes `LP_DEAD`, a stub waiting for `VACUUM` to clean the index entries pointing to it.

BEFORE pruning	AFTER pruning
<code>idx → lp1 (dead, root)</code>	<code>idx → lp1 [LP_REDIRECT → lp3]</code>
<code>lp2 (dead, heap-only)</code>	<code>lp2 [unused: space reclaimed]</code>
<code>lp3 (live, heap-only)</code>	<code>lp3 (live, heap-only)</code>

Making HOT Work for You

- Lower a table's `fillfactor` (e.g., 80–90) for update-heavy tables so each page keeps room for new versions of its own rows.
- Don't index columns you update constantly unless you need to.
- Watch the counters: `pg_stat_user_tables.n_tup_hot_upd` versus `n_tup_upd` tells you what fraction of updates are HOT.

```
SELECT relname, n_tup_upd, n_tup_hot_upd,
       round(100.0 * n_tup_hot_upd / nullif(n_tup_upd, 0), 1) AS hot_pct
FROM   pg_stat_user_tables
ORDER BY n_tup_upd DESC
LIMIT 10;
```

Trap

A long-running transaction holding back the cleanup horizon (Section 3.3) prevents pruning from removing chain members, so the chain grows, every index lookup has to walk further, and the page eventually fills — at which point updates stop being HOT. MVCC's costs are always rooted in the same place: versions can only be reclaimed when no snapshot can see them.

3.5.3 Summary**At a glance****DELETE stamps xmax.**

The tuple stays until no snapshot can see it. Abort leaves the stamp but ignores it.

UPDATE = mark old xmax + insert new + link.

There is no in-place update. Both versions coexist; the snapshot picks which one you see.

Without HOT, every index gets a new entry.

A new TID means new index entries for every index — heap and index bloat together. That is the cost Chapter 2 already priced from the leaf side.

Row locks also live in xmax.

LOCK_ONLY / MultiXact; locking a row dirties its page.

HOT keeps the index on the chain root.

No indexed column changed and the page has room. Page pruning reclaims heap-only dead versions without a full vacuum — until the cleanup horizon holds the chain open.

3.6 Chapter Summary

Five topics, one model: writers append versions, readers pick among them with a snapshot, and indexes never decide visibility. Chapter 2 left you with `key → ctid`; this chapter answers what happens after you follow that pointer.

MVCC in one picture

```

Writer:  never overwrite → stamp xmax on old, insert new, link ctid
Reader:  take snapshot (xmin:xmax:xip[]) → ask heap header
Index:   (key, TID) only → heap (or VM) decides visible to *you*
Horizon: lowest published snapshot xmin → VACUUM's cleanup limit
HOT:     same-page, no indexed cols → index stays on chain root

```

Most important

3.1: Versions beat locks for read–write.

Strict 2PL makes readers and writers take turns on one copy (S vs X, deadlocks, lock footprint). MVCC gives the reader an old version and the writer a new one. Write–write on the same row still blocks.

3.2: The header names; `pg_xact` judges.

`xmin/xmax` identify creator and deleter. `pg_xact (clog)` is the commit-status source of truth. Hint bits cache that answer on the tuple so later readers skip the commit log. `cmin/cmax` order statements inside one transaction (Halloween problem). `ctid` links old → new.

3.3: A snapshot is not a data copy.

`xmin:xmax:xip[]` says who was still running. `xip` exists because XID order $\neq$ commit order. Read committed retakes per statement; repeatable read / serializable take one at the first query. The lowest published snapshot `xmin` is the cluster-wide cleanup horizon — one long transaction bloats every table.

3.4: Insert happened and delete has not.

`HeapTupleSatisfiesMVCC` answers those two questions against *your* snapshot. Indexes carry no visibility; every scan reaches the heap or the visibility map. Concurrent writers: wait if still running; on commit, read com-

mitted re-checks the newest version, repeatable read raises a serialization failure.

3.5: Update is versioning; HOT is the opt.

DELETE stamps `xmax`. UPDATE marks the old version, inserts a new one, and links via `ctid`. Without HOT every index grows. HOT keeps the index on the chain root when no indexed column changes and the page has room; page pruning reclaims heap-only dead versions without a full vacuum.

The previous chapter's contract still holds.

An index maps a key to a `ctid`. This chapter is what you do after you follow it — and why an index-only scan still needs the visibility map (Section 2.2).

Bloat has one root cause.

Versions can only be removed when no snapshot can still see them. Horizon lag, missing HOT room, and indexes on hot columns all amplify that same fact.

What this chapter has not covered yet.

Row/table/advisory locks in detail, deadlock detection, isolation-level deep dives, SSI predicate locks, and XID wraparound / freezing — those follow as the concurrency half of the chapter.