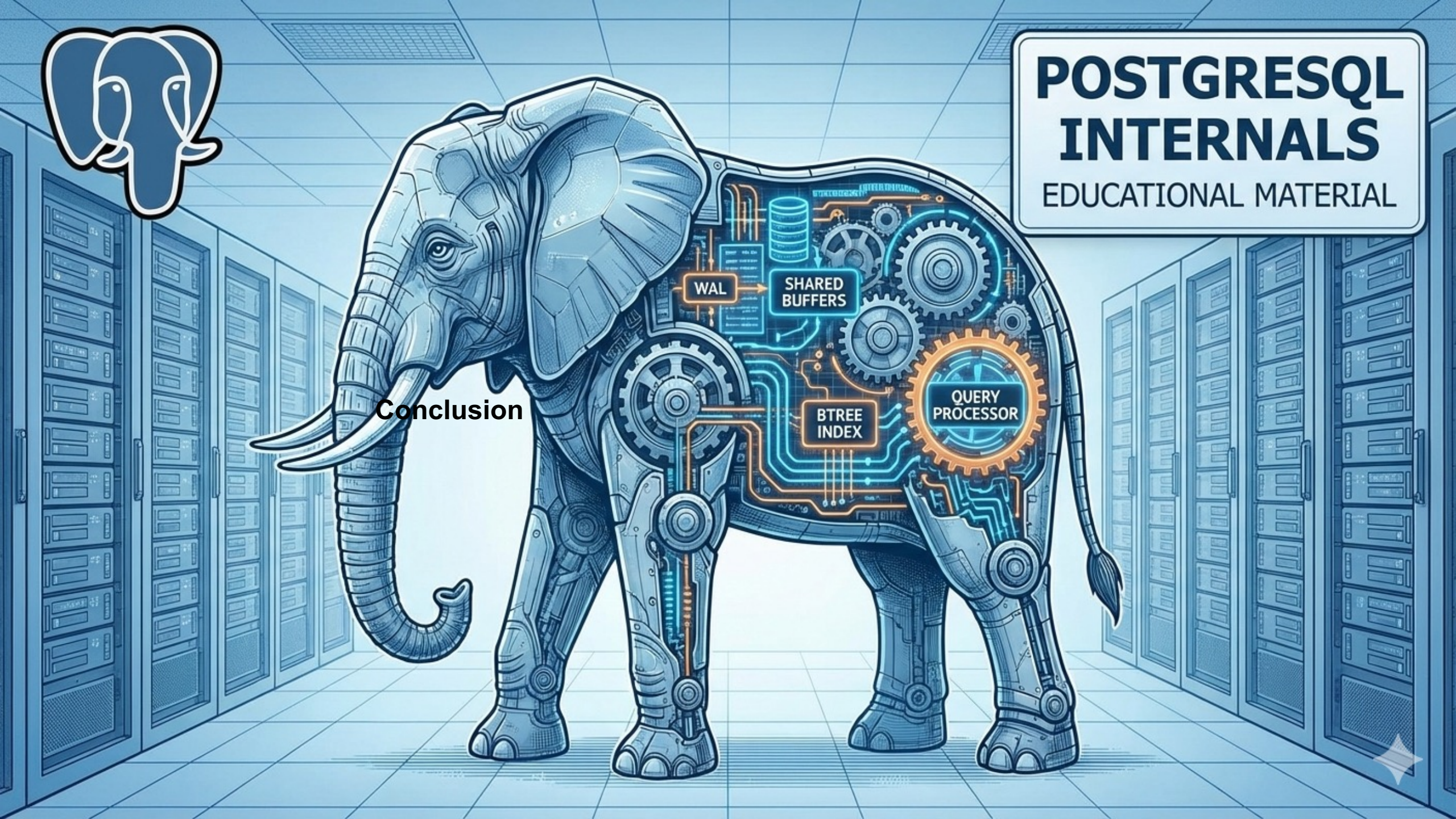
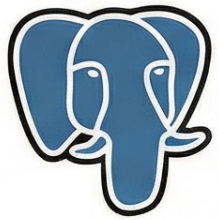

POSTGRESQL
INTERNALS
EDUCATIONAL MATERIAL
WAL
SHARED BUFFERS
BTREE INDEX
QUERY PROCESSOR
Conclusion

Mastering the Data Ecosystem: Profile of a PostgreSQL Expert & Technology Evangelist

Core Expertise: The PostgreSQL Specialist







Migration Specialist: Legacy to PostgreSQL

Expert at transitioning high-cost legacy databases (specifically Oracle) to PostgreSQL with minimal downtime and risk.



High Availability (HA) Architecture

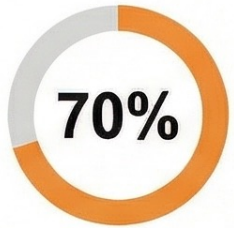
Designs "always-on" infrastructures using EDB BDR to ensure 99.999% availability and global business continuity.



Enterprise Performance Advisory

Optimizes complex database environments to reduce latency and establish enterprise-grode operational best practices.

25 Years of Quantifiable Impact



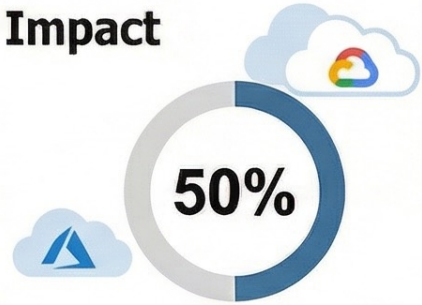
Improvement via SQL Tuning

Achieved a 70% increase in overall database performance for clients like Rockplace through continuous SQL optimization.



Performance Boost in Gaming

Led database performance efforts at Supercat for "The Kingdom of the Winds: Yoon," resulting in a 60% overall gain.



Faster Batch Processing

Improved batch job performance by 50% for major financial institutions like Shinhan Bank.

Performance Benchmarks (Specific Gains across Sectors)		
EnterpriseDB	PostgreSQL	99.999% Availability Architecture
Rockplace	PostgreSQL	70% SQL Performance Improvement
Supercat Inc	MongoDB/MariaDB	60% Increase in DB Performance
WiderPlanet	AWS RedshiR/BigQuery	60% Increase in Ad Delivery Efficiency
UgensSNC	Oracle	50% Increase in Card Payment Performance
KDMedia	SQL Server	40% Workflow Streamlining via Automation

Technology Evangelism & Instruction





Proven Instructor & Speaker

Delivered keynotes and technical sessions at SECaaS 2023, Devfest Songdo, and Google I/D Extended Seoui on topics ranging from Graph DBs to Neural Networks.



Diverse Certification Portfolio

Holds professional certifications in Snowflake, AWS Solutions Architect, TigerGraph, Oracle 11g, and Couchbase.



Content Curator & Translator

Experienced in beta-reading and translating complex IT terminology for books to ensure clarity for diverse technical audiences.

Reference Material & Contact

Professional Connections

82-10-8948-9592

jazzlian@gmail.com

Digital Footprint

www.ninanolabs.com

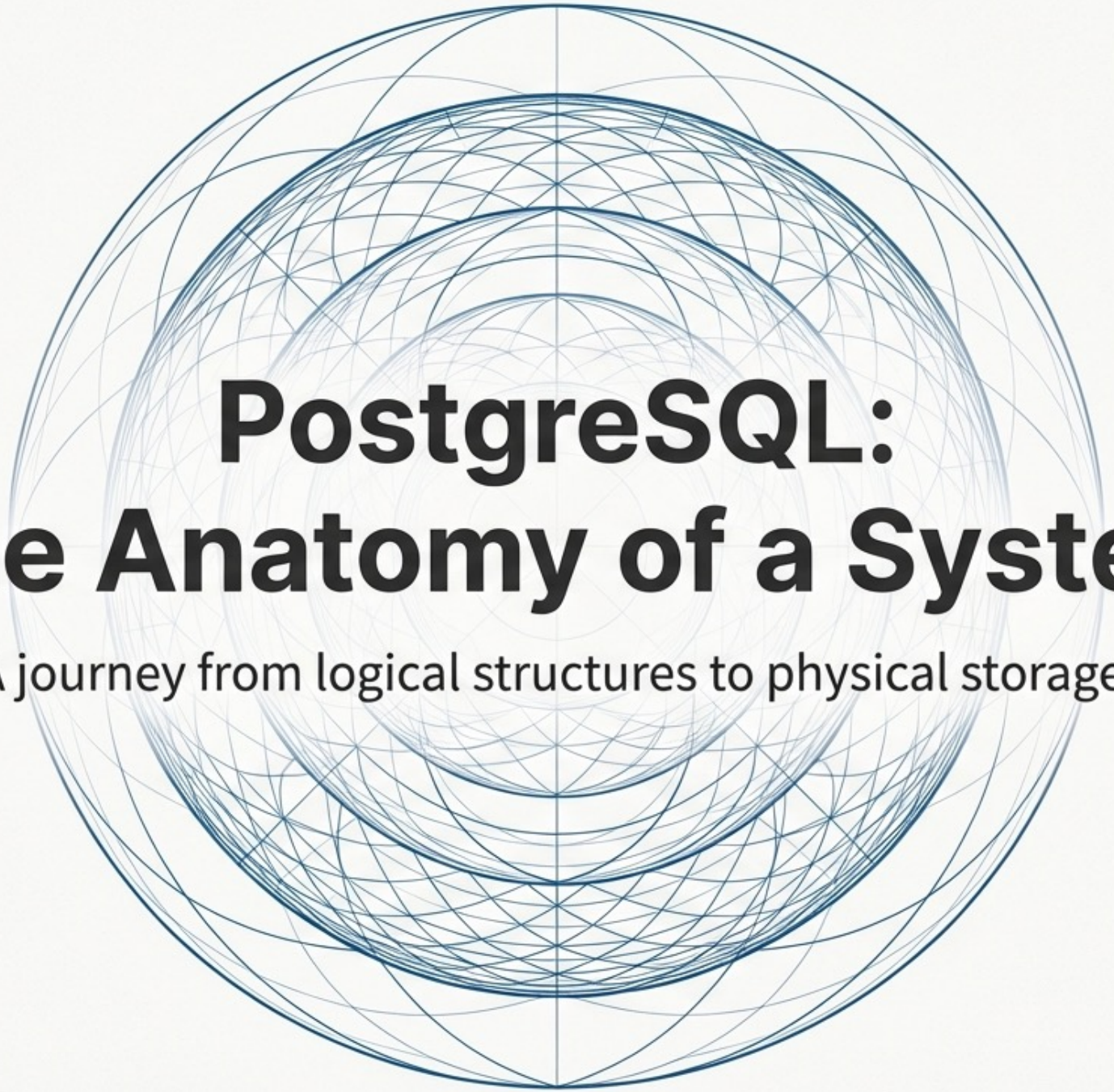
linkedin.com/in/myoungsig-youn

Beyond the Data

A jazz enthusiast and fan of Haruki Murakami, bringing a creative depth to the world of technical architecture.

NotebookLM

1. Introduction



PostgreSQL: The Anatomy of a System

A journey from logical structures to physical storage.

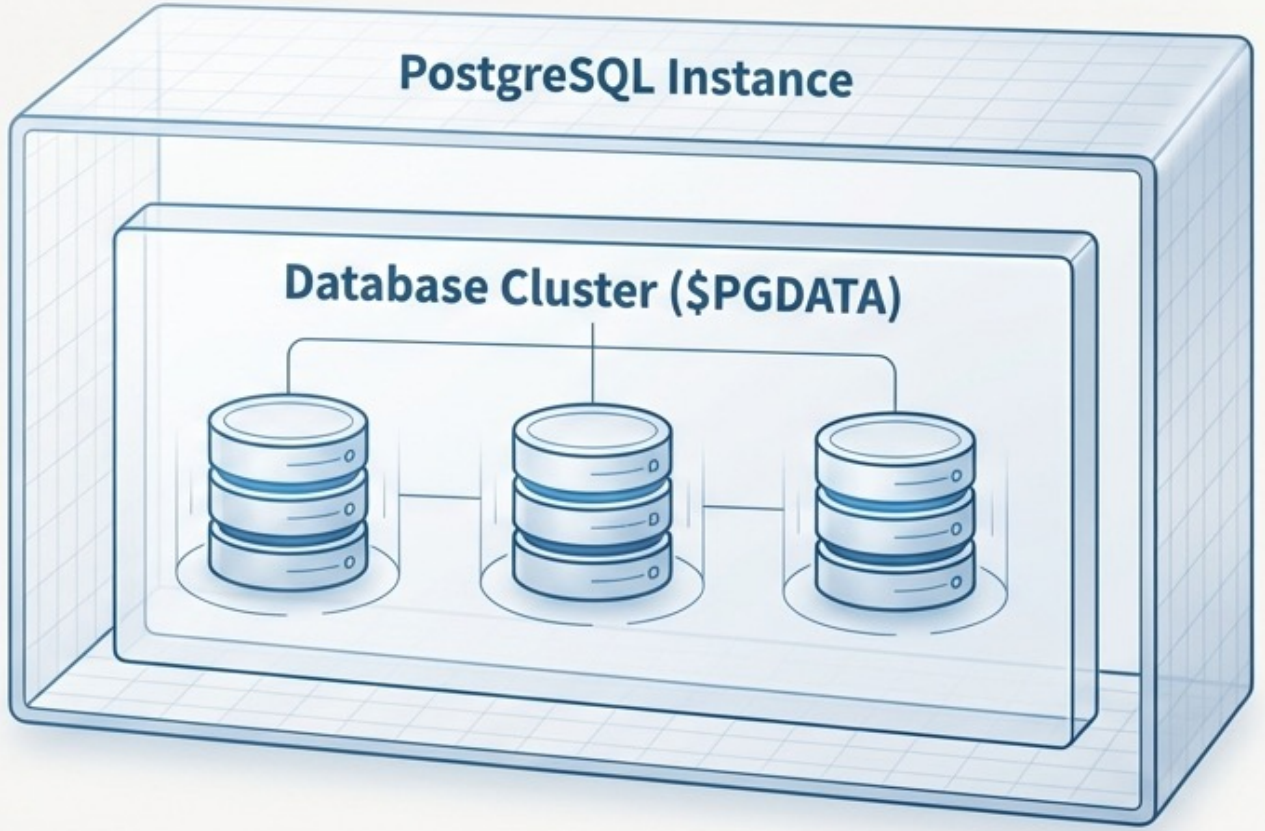
NotebookLM

It All Starts with a PostgreSQL Instance

A running PostgreSQL server is called an **instance**. This instance manages a collection of databases known as a "database cluster."

All files related to a cluster are stored in a single main folder, the "data directory."

This directory is commonly referred to by its environment variable name: `PGDATA`.



Understanding the `PGDATA` directory is the first step in physical database administration, from backups to replication.

NotebookLM

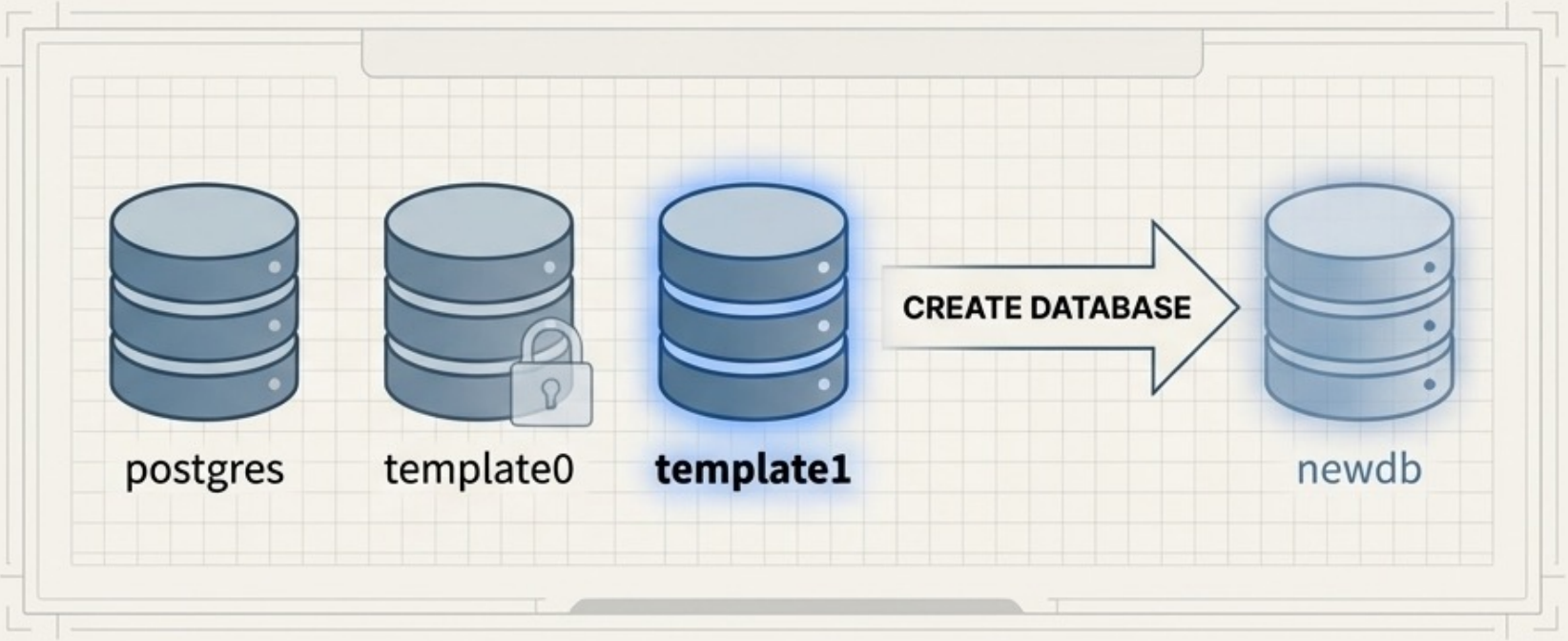
Every New Database Begins as a Clone

- When you initialize a cluster, PostgreSQL creates three default databases. New databases are created by copying a template.

`template0`: A pristine, unchangeable template. Used for special cases like restoring from a backup. It should never be modified.

`template1`: The default template. Any objects you create in template1 will appear in all subsequent new databases.

`postgres`: A standard, general-purpose database for users and tools to connect to initially.



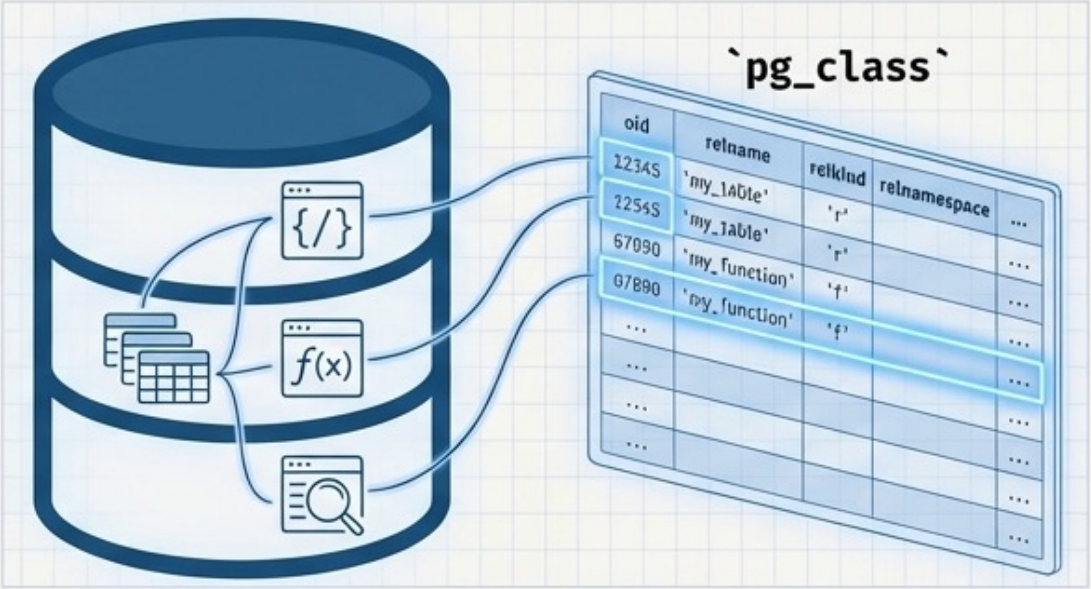
Customizing template1 can pre-load all new databases with specific extensions, functions, or schemas, standardizing your environment.

NotebookLM

The System's Internal Map: System Catalogs

All metadata about database objects (tables, indexes, functions, etc.) is stored in a special set of tables called the system catalog.

- Each database has its own set of catalog tables.
- Catalog table names begin with `pg_`, like `pg_database`.
- The primary key is often an `oid` (Object Identifier), a cluster-wide unique integer.




```
SELECT datname, oid FROM pg_database;
```

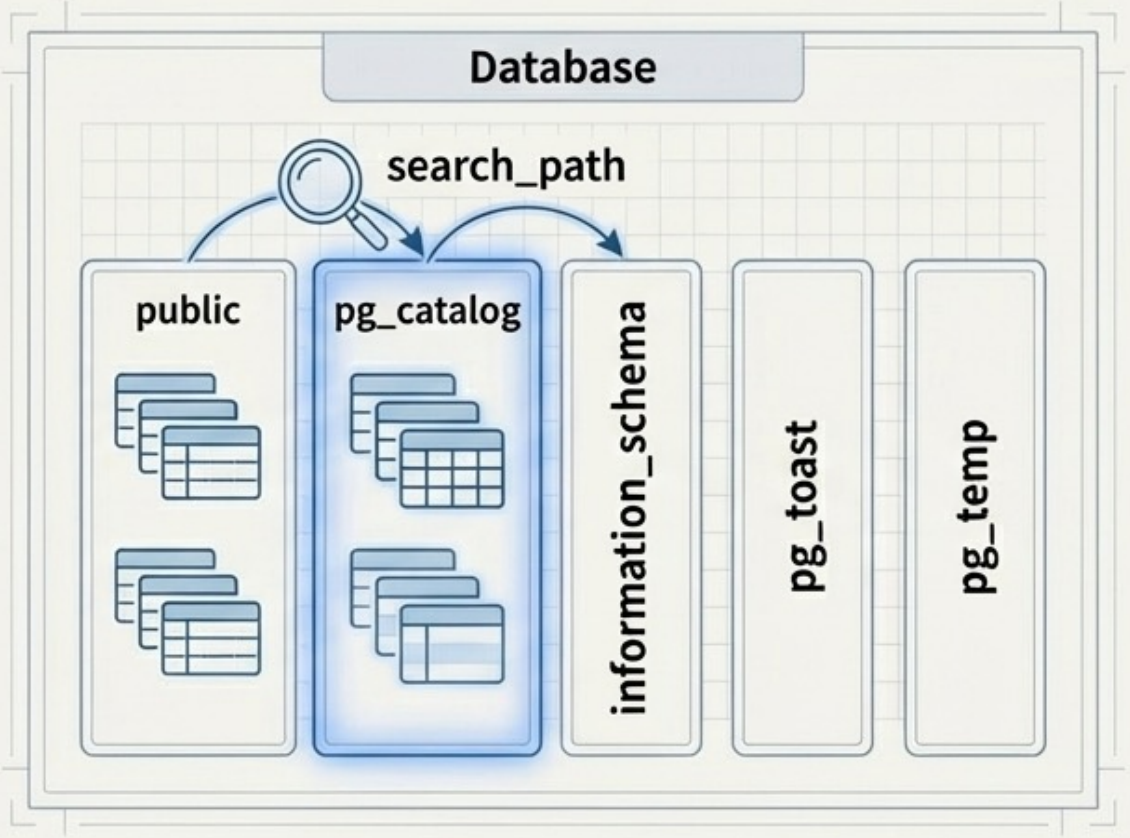
You can query the system catalog directly with SQL to gain deep insights into your database structure. Tools like `psql` use these catalogs for commands like `\d`.

NotebookLM

Organizing Objects Within a Database: Schemas

Schemas are namespaces that contain all database objects (tables, views, etc.). They allow objects with the same name to coexist in different schemas.

- The `search_path` parameter determines the order in which PostgreSQL looks for an object when a schema is not specified. `pg_catalog` is always implicitly included.



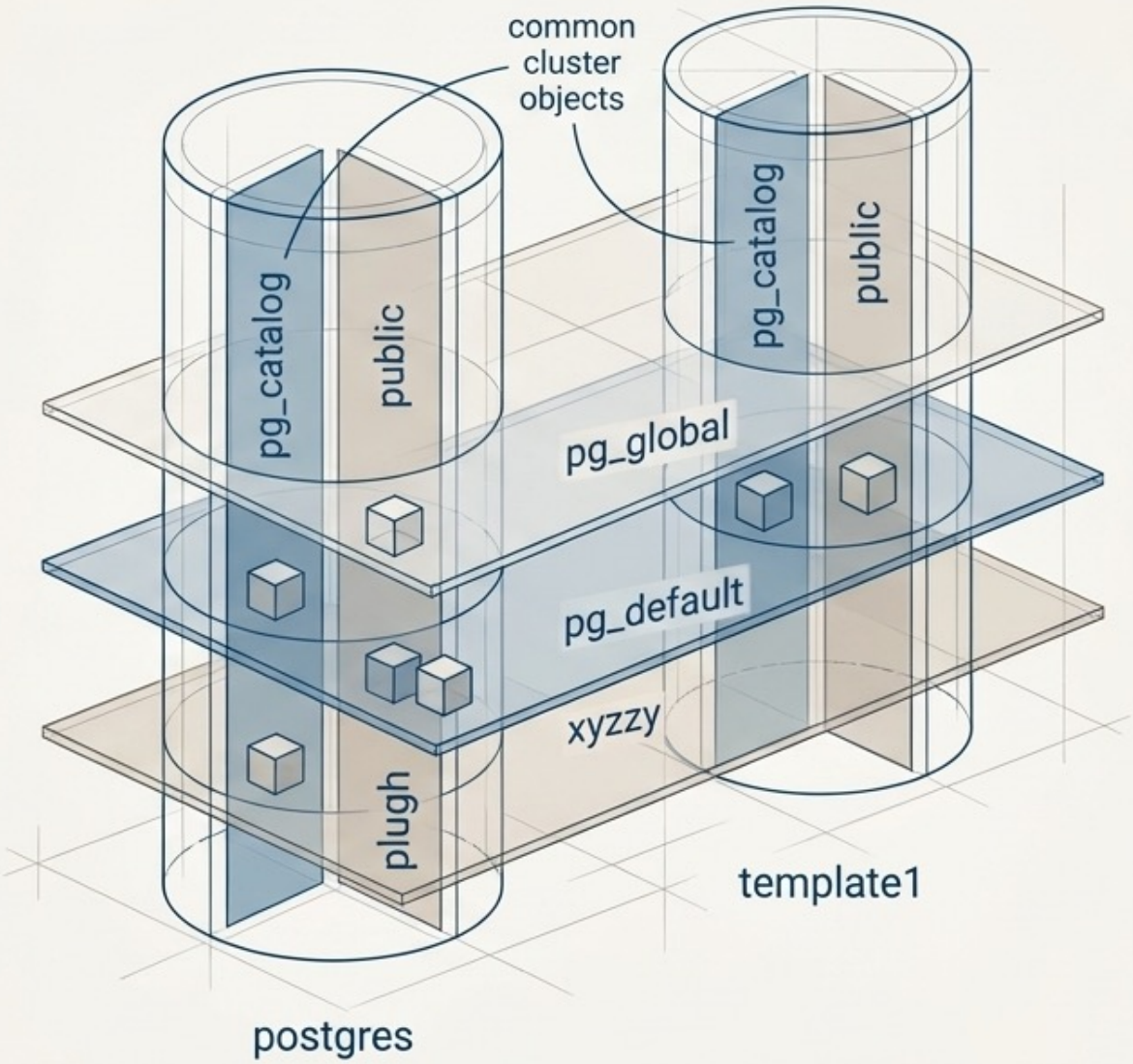
Using custom schemas is a powerful way to organize your application's data, manage permissions, and avoid naming conflicts in complex systems.

NotebookLM

Mapping Logical Data to Physical Disks: Tablespaces

While databases and schemas organize data logically, tablespaces define where that data data is physically stored on the file system. A tablespace is essentially a directory.

- Tablespaces allow distributing data across different storage devices (e.g., fast SSDs for tables, slower disks for archives).
- The logical structure (databases, schemas) and physical placement (tablespaces) are independent.
- Default Tablespaces
  - `pg_default`: Located in `$PGDATA/base`
  - `pg_global`: Located in `$PGDATA/global`

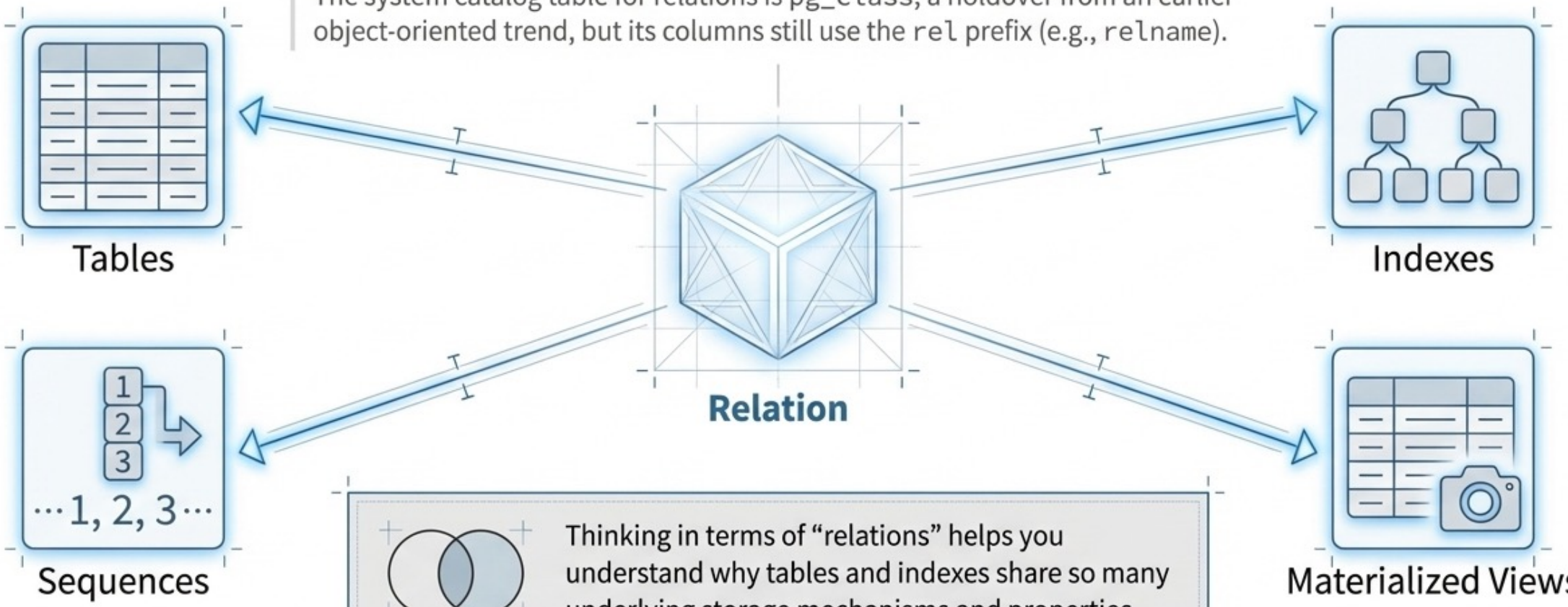


NotebookLM

The Universal Object is the ‘Relation’.

In PostgreSQL, objects composed of rows are generically called “relations.” This is a unifying concept for several distinct object types.

> The system catalog table for relations is `pg_class`, a holdover from an earlier object-oriented trend, but its columns still use the `rel` prefix (e.g., `relname`).



Thinking in terms of “relations” helps you understand why tables and indexes share so many underlying storage mechanisms and properties.

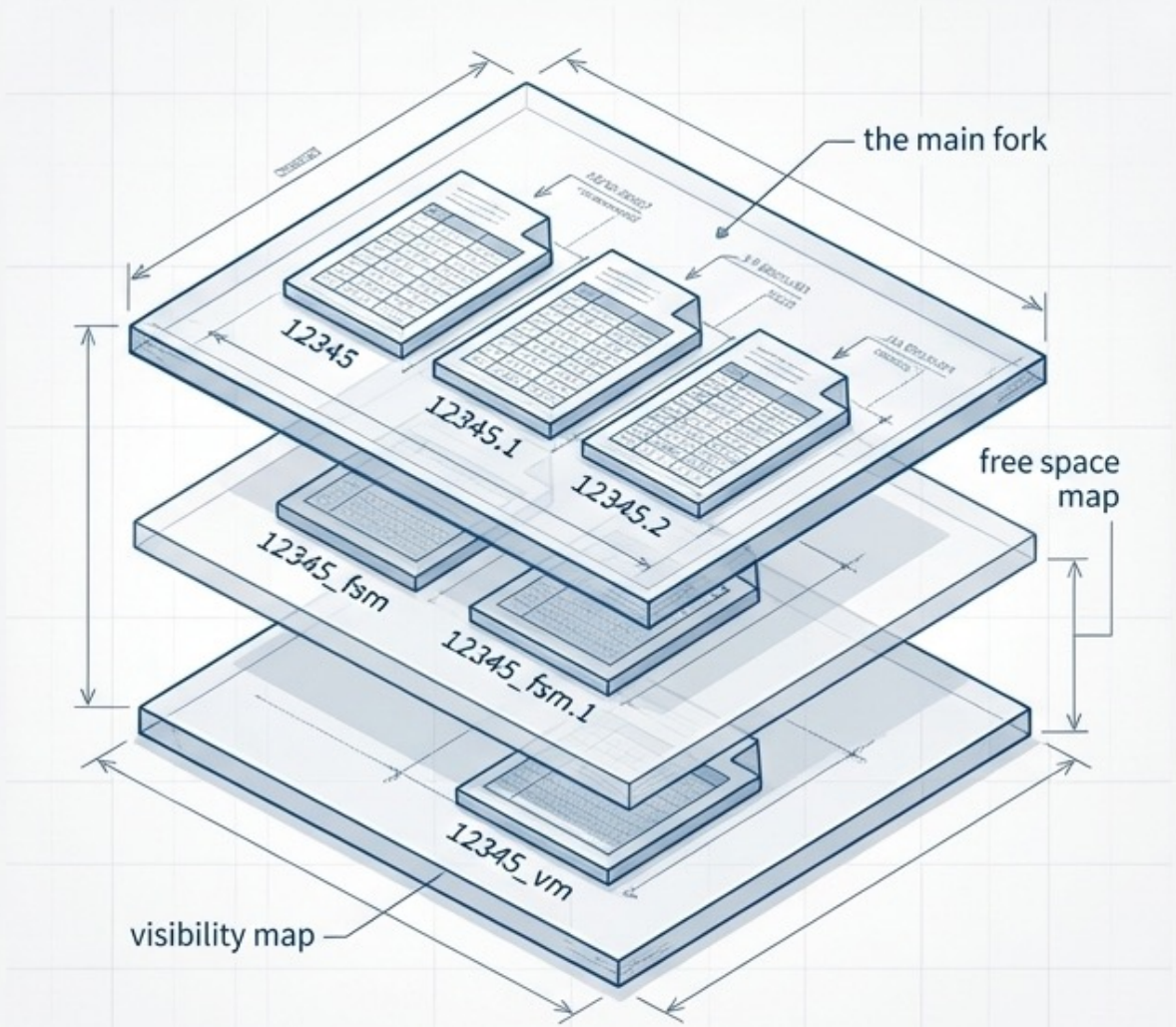
NotebookLM

How a Relation Becomes a Set of Files.

Every relation is stored on disk not as a single file, but as multiple files called “forks.” Each fork stores a specific type of data associated with the relation.

Physical File Structure

- **File Naming:** The base filename is the `relfilenode` OID (e.g., `12345`).
- **Forks:** Each fork has a suffix (e.g., `_fsm`, `_vm`).
- **Segmentation**: If a file exceeds 1 GB, it is split into segments (e.g., `12345.1`, `12345.2`).



NotebookLM

The Files Behind Every Table.

A simple, unlogged table has at least three types of files (forks) on disk, each with a critical function.

Main Fork (16385)

Contains the actual table or index data (the rows or tuples).



Initialization Fork (_init)

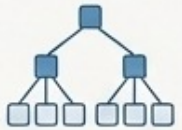
Exists only for UNLOGGED tables. A blank copy used to reset the table after a server crash.





Table Relation

Free Space Map (_fsm)



A tree-structured map that tracks available space within pages, allowing INSERT operations to quickly find room.

Visibility Map (_vm)



A bitmap with two bits per page.

Bit 1: Is the page 'all-visible'? (Allows Index-Only Scans).
Bit 2: Is the page 'all-frozen'? (Allows VACUUM to skip it).

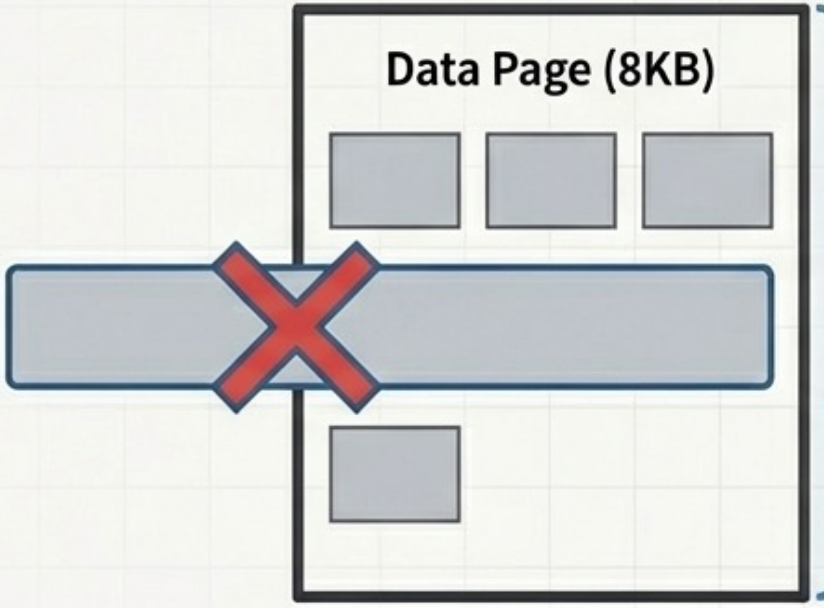


The existence of the FSM and VM is critical for PostgreSQL's performance. They allow **VACUUM** and **INSERTs** to operate efficiently without scanning the entire table.

NotebookLM

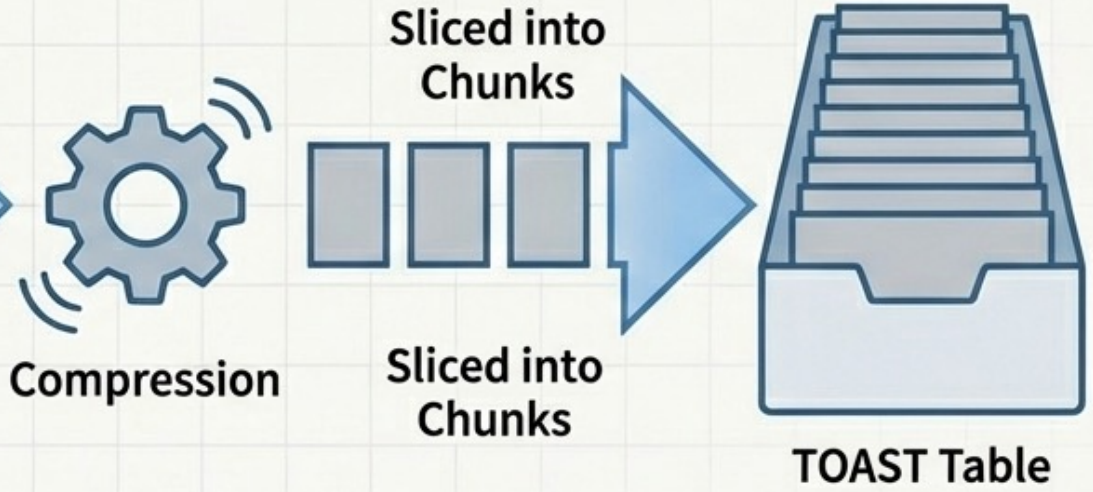
What Happens When a Row Is Too Big for a Page?

Stage 1: The Problem



A single row in a table must fit within a single data page.

Stage 2: The Solution: TOAST



TOAST works transparently, but it's not free. Compressing and fetching out-of-line data adds overhead. This is a key reason to avoid `SELECT *` on tables with large text or json columns.

NotebookLM

The Four Storage Strategies of TOAST.

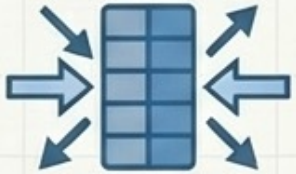
You can control the TOAST behavior of a column by setting its storage strategy. The default depends on the data type.

PLAIN



Disables TOASTing. For fixed-size types like `integer`.

EXTENDED



Allows compression and out-of-line storage. Default for `text`, `json`.

EXTERNAL

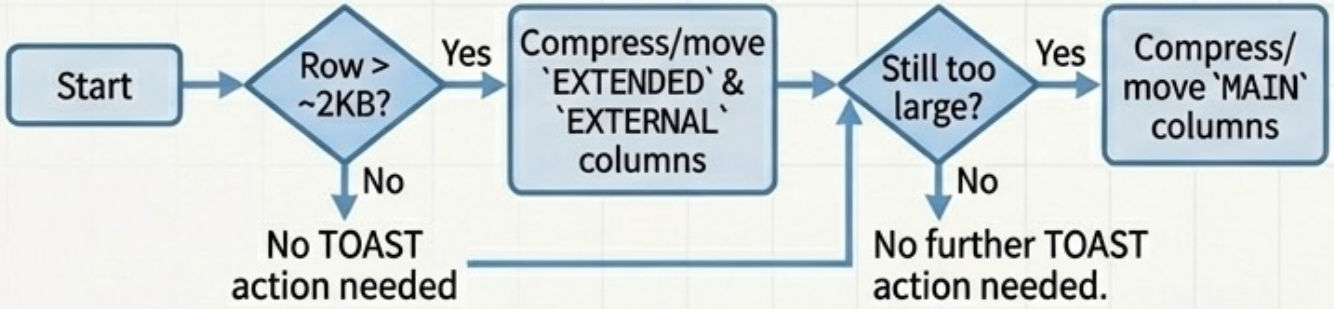


Disables compression but allows out-of-line storage. Good for JPEGs.

MAIN



Allows compression, but out-of-line is a last resort.




```
-- Check storage
SELECT attname, attstorage FROM pg_attribute
WHERE attrelid = 't'::regclass;

-- Change strategy
ALTER TABLE t ALTER COLUMN d SET STORAGE external;
```

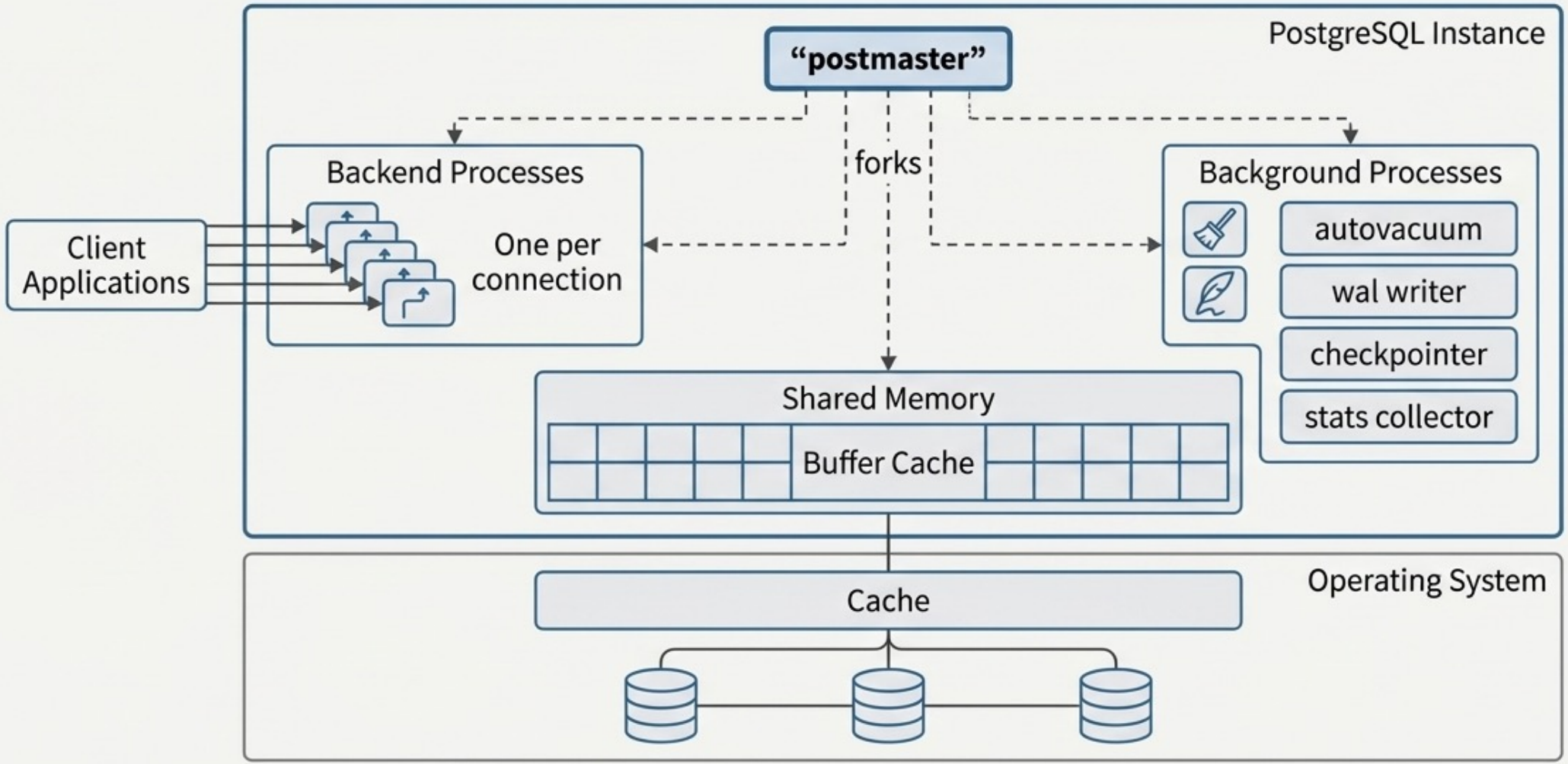
CPU

Explicitly setting a column's storage to `EXTERNAL` for inherently compressed data can prevent useless compression attempts and improve write performance.

NotebookLM

The Living System: A Master Process and its Workers

- A PostgreSQL instance is a collection of cooperating processes managed by a single parent process, traditionally called the `postmaster`.



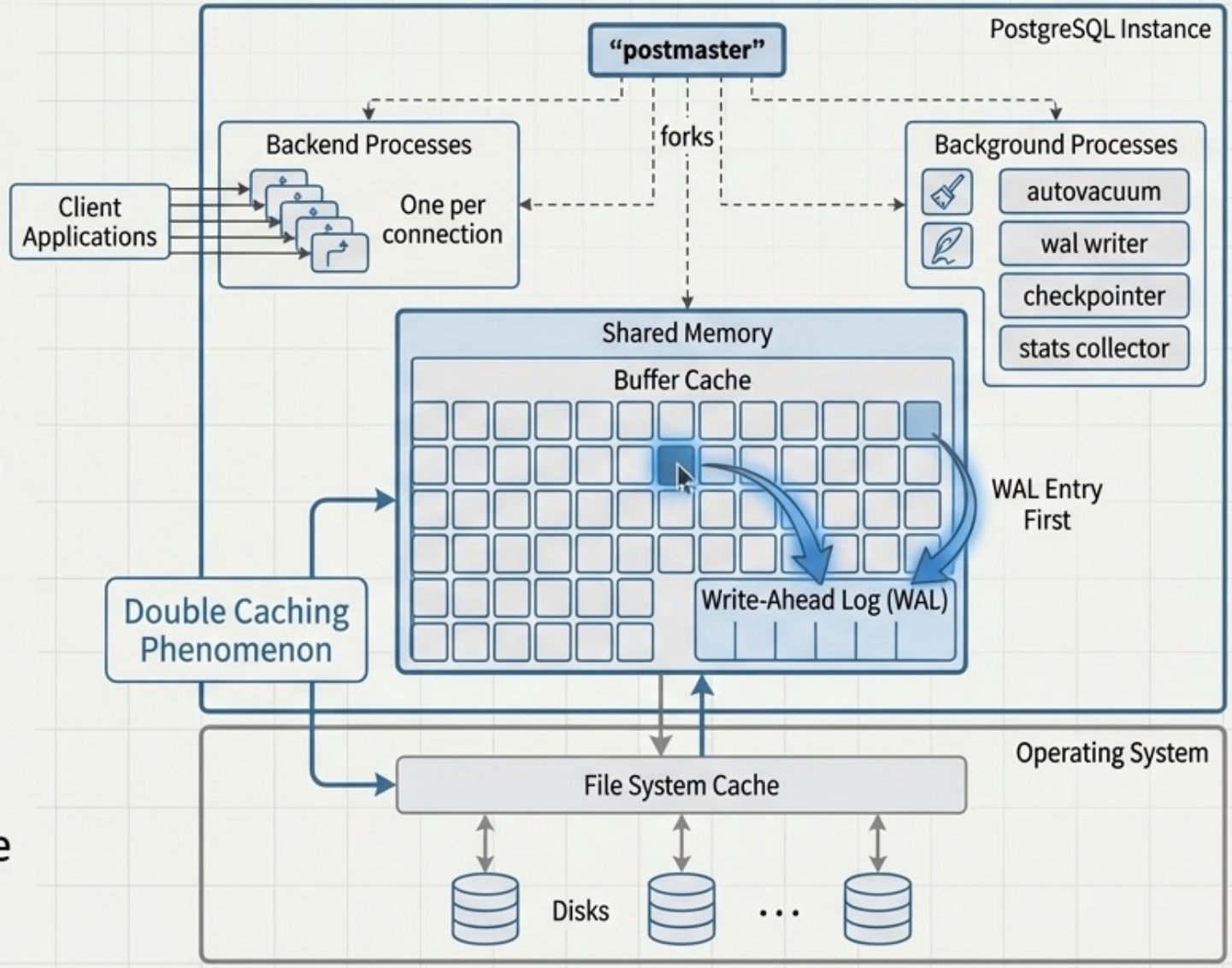
NotebookLM

The Central Nervous System: Shared Memory and Caching.

To overcome slow disk I/O, PostgreSQL relies heavily on caching data in RAM. The `postmaster` allocates a large region of shared memory accessible by all its child processes.

Key Components:

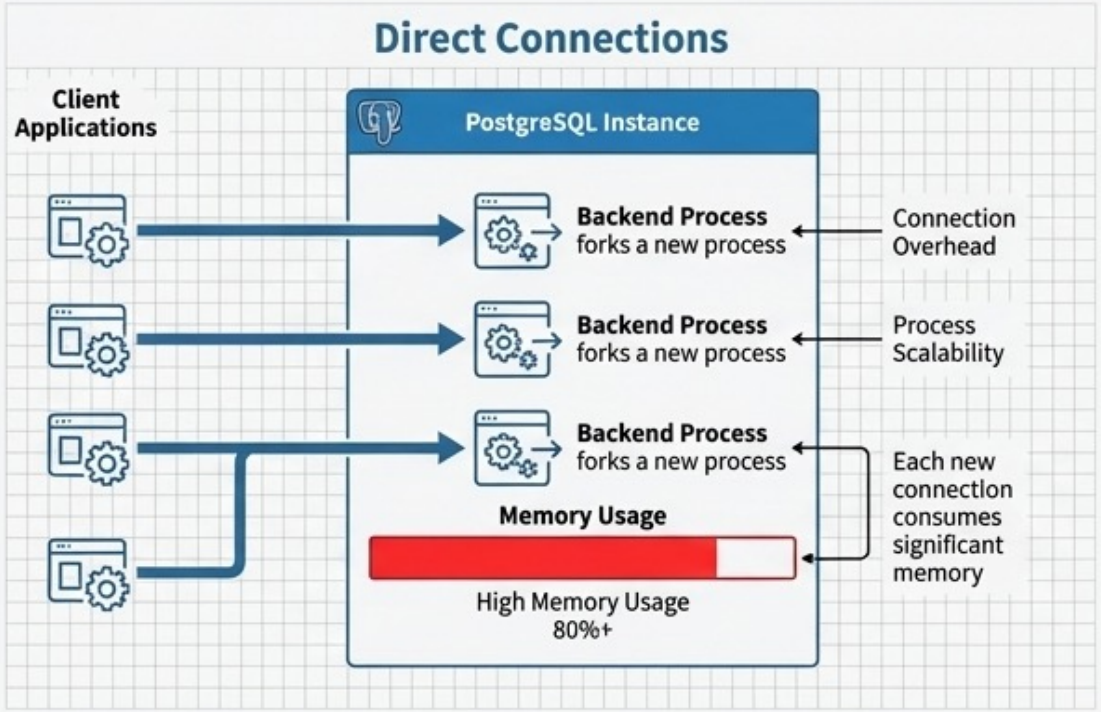
- **Buffer Cache:** The largest part of shared memory. It holds recently used 8KB data pages.
- **Write-Ahead Log (WAL):** To ensure durability, changes are first written as a log entry in the WAL *before* the buffer cache is modified.
- **Double Caching:** Data can exist in both PostgreSQL's buffer cache and the OS file system cache.

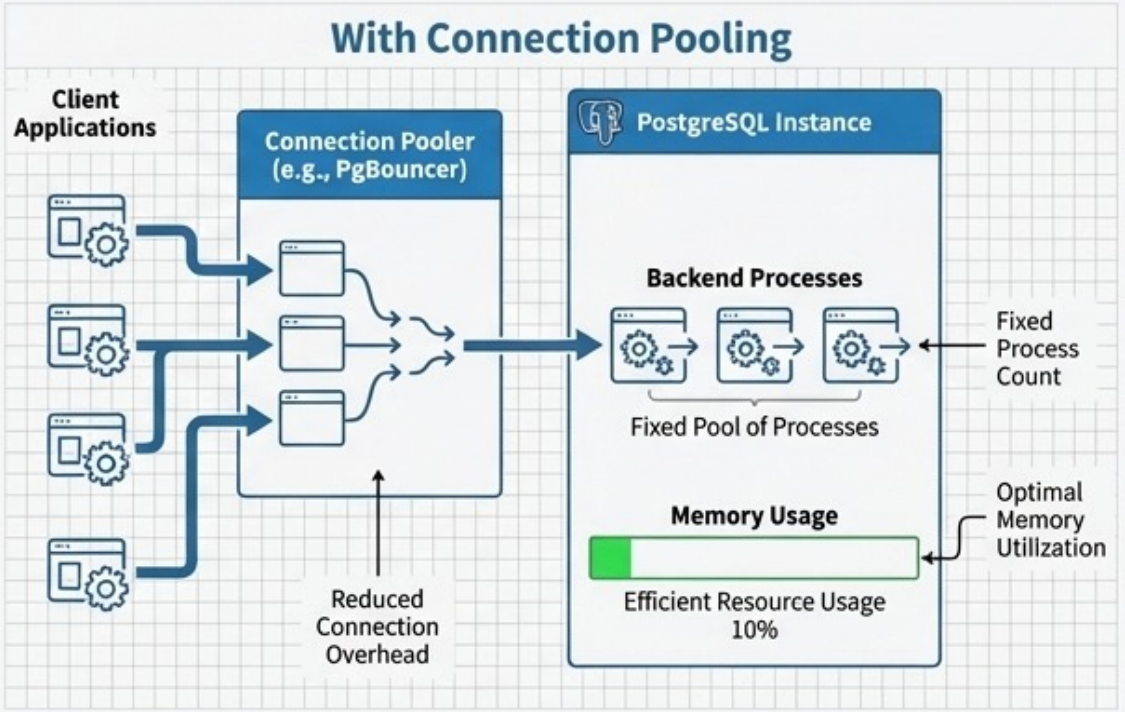


NotebookLM

The Hidden Cost of a Connection

Forking a dedicated backend process for every client connection is robust but has performance drawbacks, especially with many short-lived connections.





For any serious production workload, a connection pooler is not optional—it is a mandatory component of the architecture to ensure performance and stability.

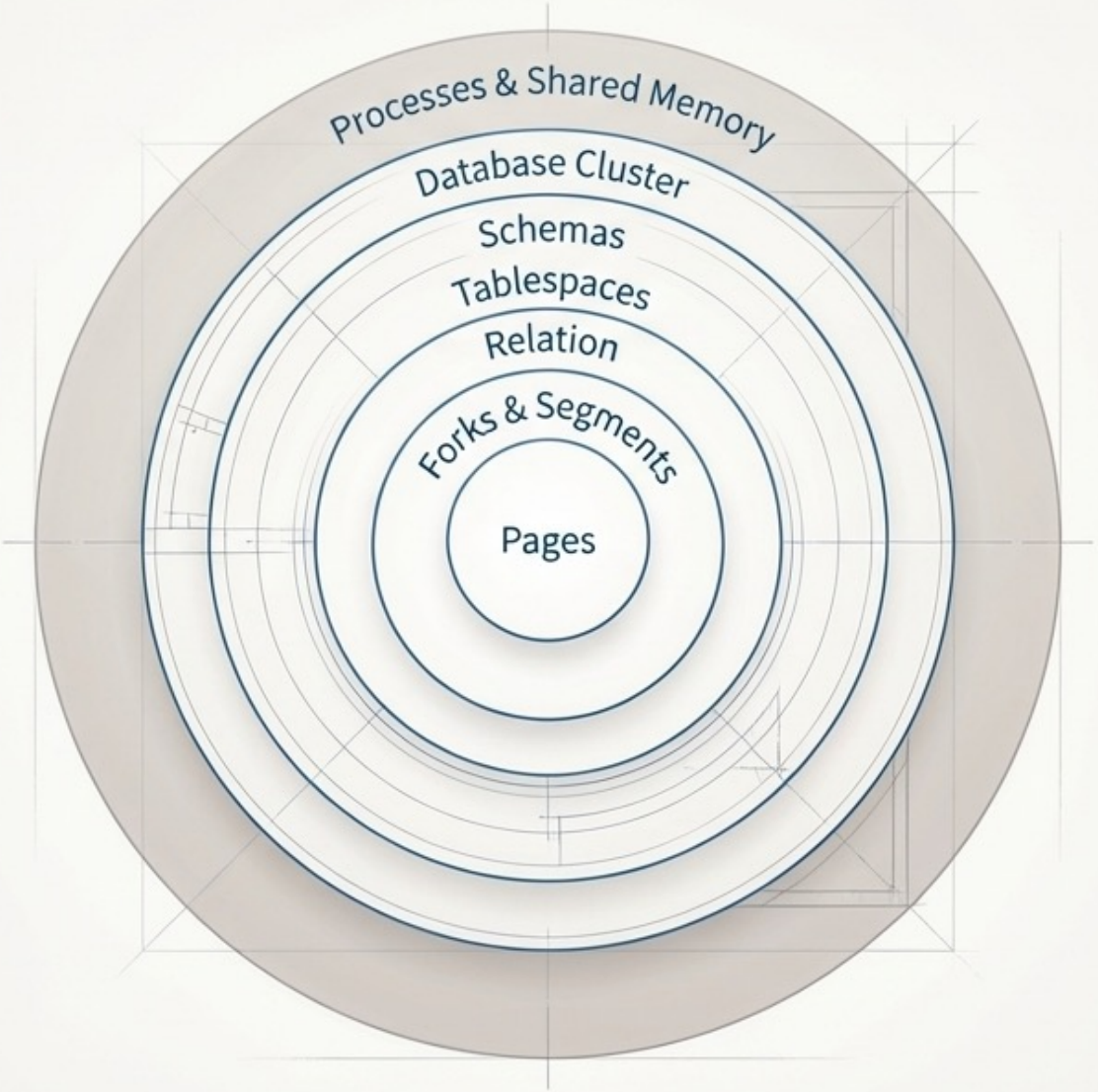
NotebookLM

From Logical Abstraction to Physical Reality

Summary of the Journey

We've dissected the PostgreSQL system, peeling back the layers from the abstract to the concrete:

- A logical **Database Cluster**...
- ...organized by **Schemas**...
- ...is mapped to physical disks via **Tablespaces**...
- ...where every **Relation**...
- ...is stored as a set of **Forks** and **Segments**...
- ...composed of 8KB **Pages**.
- This entire data structure is brought to life by a system of **Processes** communicating through **Shared Memory**.



Final Takeaway

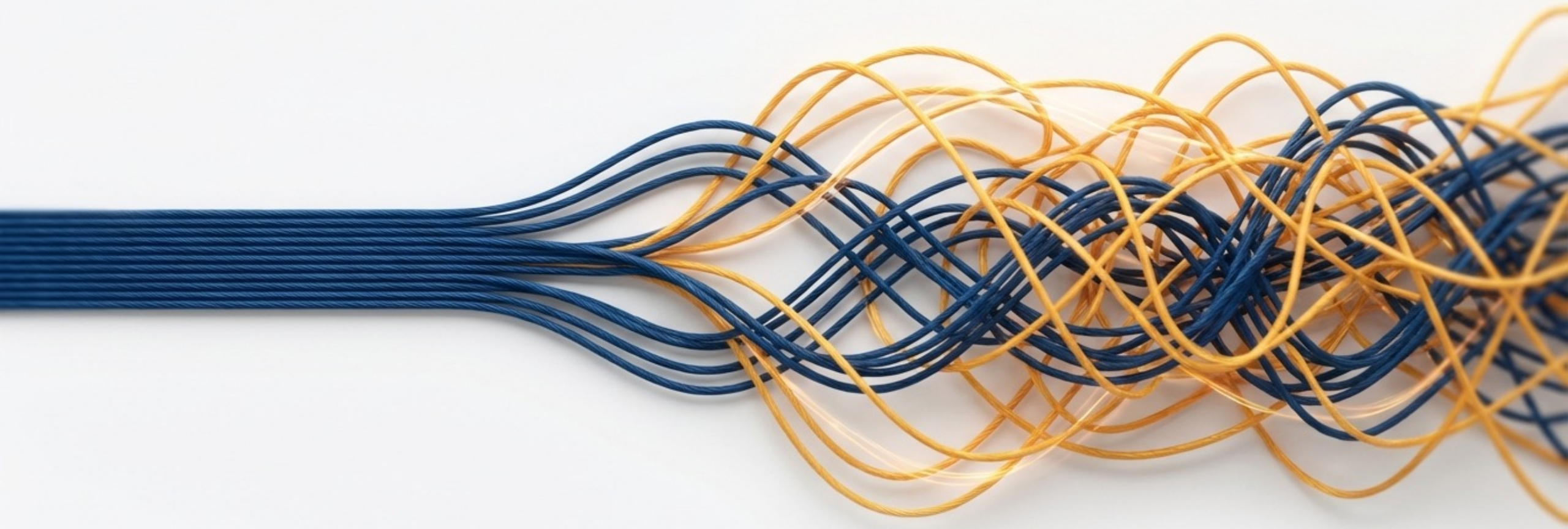
Every choice you make—from schema design to storage provisioning to connection strategy—has a direct impact on these underlying components. A clear mental model of this anatomy is the most powerful tool for building fast, reliable, and scalable systems on PostgreSQL.

NotebookLM

2. Isolation

Demystifying Database Isolation

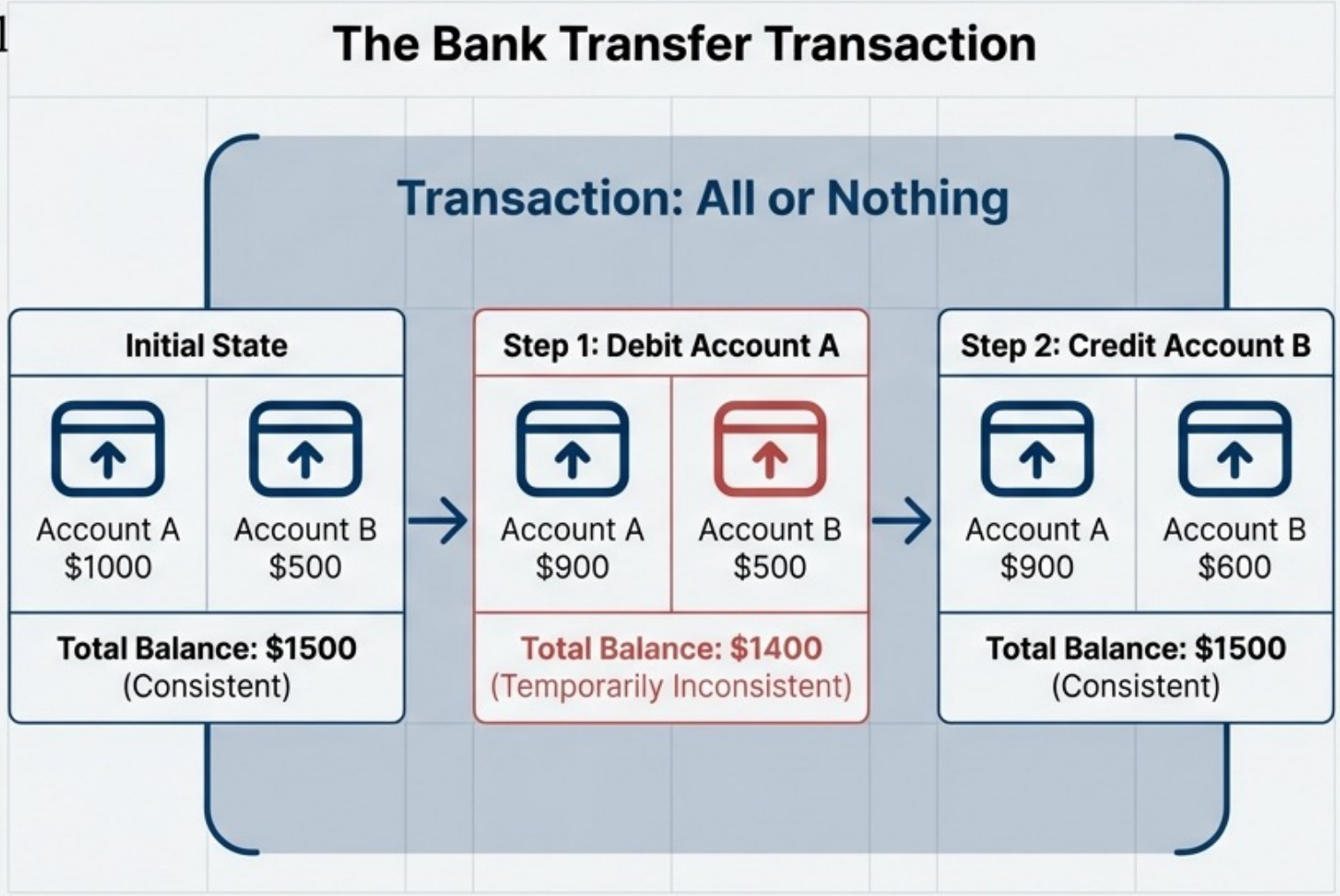
A Deep Dive into Concurrency, Anomalies, and PostgreSQL's Approach



NotebookLM

The Quest for Consistency: Data Must Be Correct

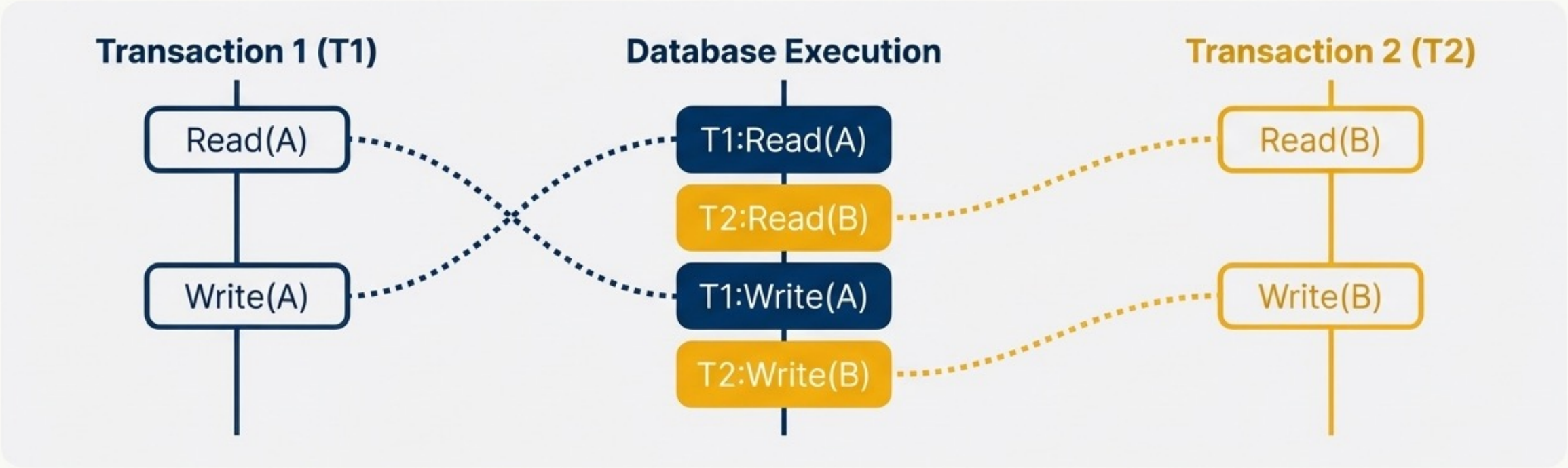
The primary function of a relational database is maintaining data consistency. This is different from data integrity, which is enforced by database-level constraints like `NOT NULL` or `UNIQUE`. Many critical consistency rules exist at the application level; the database itself doesn't know about them. For example, the total balance across two accounts should remain constant during a transfer.



NotebookLM

The Problem: The Chaos of Concurrency

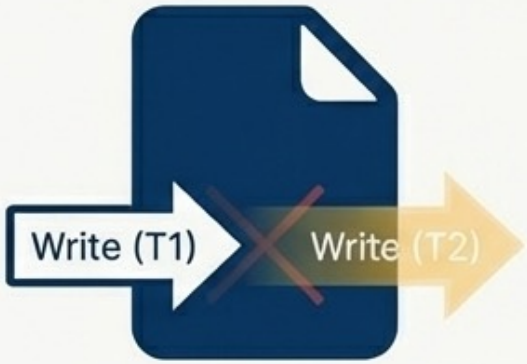
- Running transactions one by one (sequentially) would be perfectly safe, but far too slow for any modern application.
- To achieve performance, databases must execute transactions in parallel.
- This parallel execution means the operations of different transactions can get interleaved, leading to unexpected and incorrect results. These are called **concurrency anomalies.**



NotebookLM

The Rogues' Gallery of Concurrency Anomalies

The SQL standard defines isolation levels based on which of these "ghosts" they prevent from appearing in the system.



Lost Update

One transaction's committed change is overwritten by another.



Dirty Read

Reading data from a transaction that hasn't committed yet (and might roll back).



Non-Repeatable Read

Reading the same row twice in one transaction and getting different values.



Phantom Read

Re-running a query and finding new rows that have been added by another transaction.

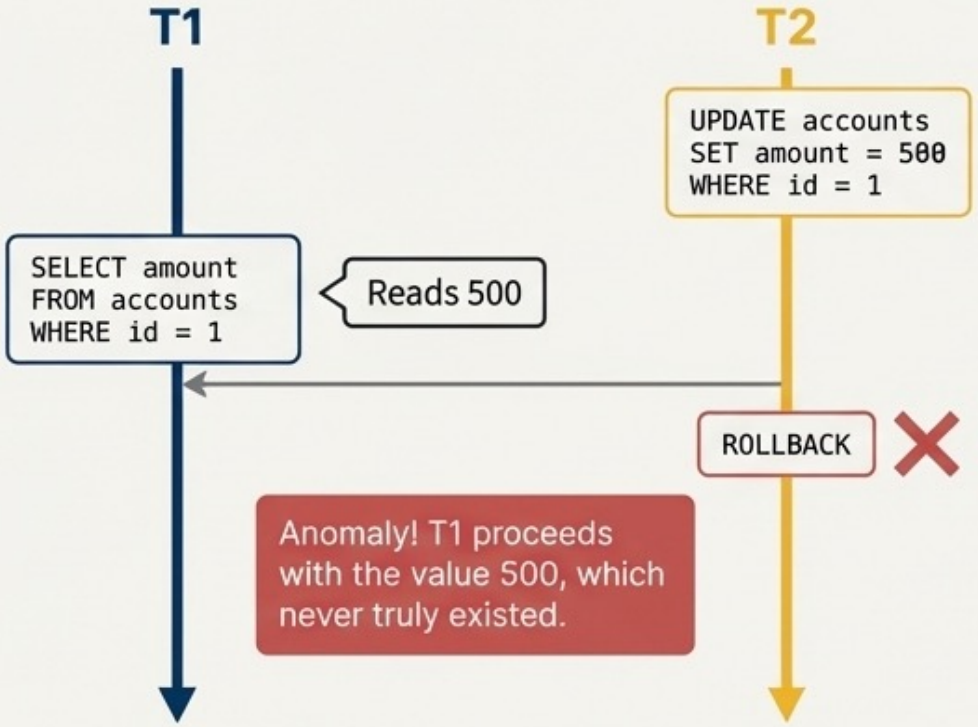
NotebookLM

Anomaly Deep Dive: Uncommitted and Changing Data

Dirty Read

Scenario: Transaction T1 reads data that T2 has modified but not yet committed. T2 then aborts (ROLLBACK).

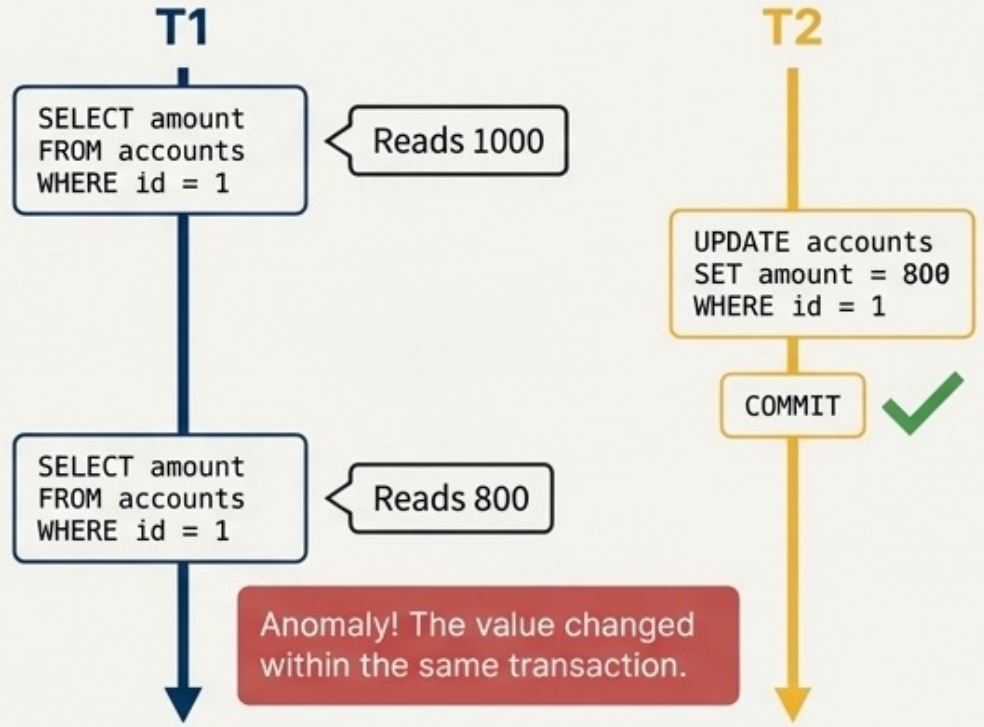
Outcome: T1 is now holding "dirty" data that never officially existed. It has seen a ghost.



Non-Repeatable Read

Scenario: T1 reads a row. T2 then updates or deletes that same row and commits its change.

Outcome: T1 reads the exact same row again later and gets a different result. The data is not “repeatable”.



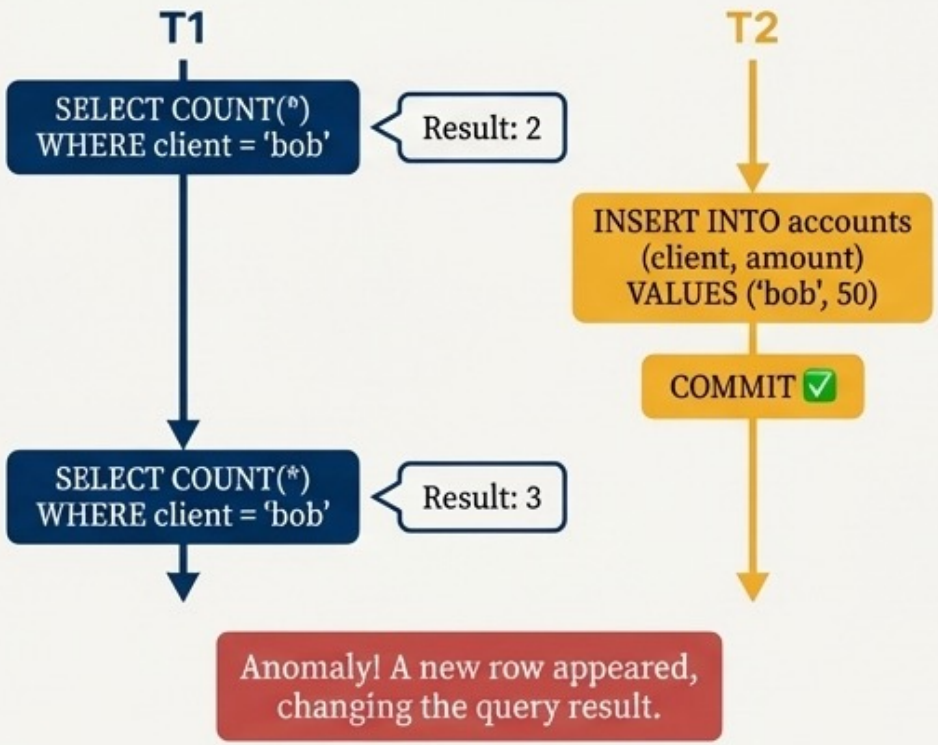
NotebookLM

Anomaly Deep Dive: Appearing Rows and Overwritten Work

Phantom Read

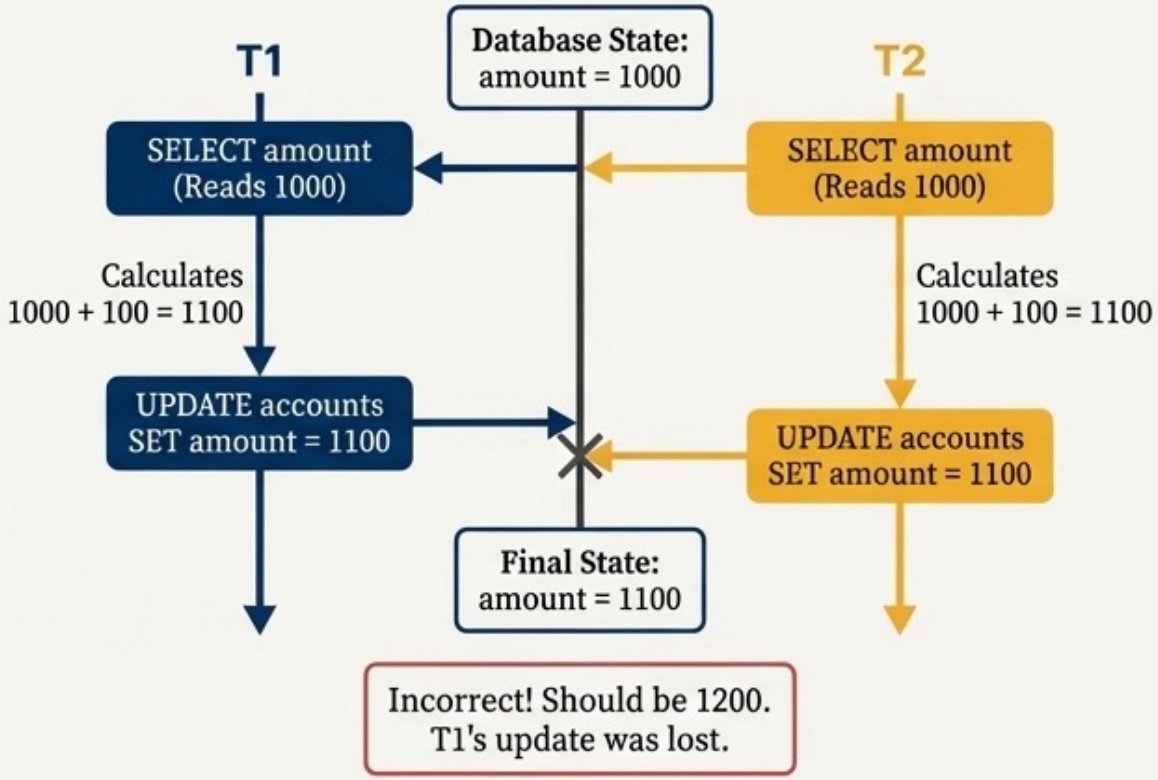
Scenario: T1 runs a query with a WHERE clause (e.g., SELECT COUNT(*)). T2 then inserts a new row that matches that WHERE clause and commits.

Outcome: T1 runs the exact same query again and gets a different number of rows. A "phantom" row has appeared.



Lost Update

Scenario: T1 and T2 both read the same value. T1 calculates a new value and writes it. T2 then calculates its own new value and writes it, overwriting T1's work.



NotebookLM

The SQL Standard: A Framework for Isolation

The SQL standard defines four isolation levels. Each level is defined by which of the concurrency anomalies it permits. The goal is to provide a trade-off between performance and consistency. The standard assumes a model based on locking, which is different from how many modern databases work.

SQL Standard Isolation Levels vs. Anomalies

Isolation Level	Lost Update	Dirty Read	Non-Repeatable Read	Phantom Read	Other Anomalies
Read Uncommitted	✓	✗	✗	✗	✗
Read Committed	✓	✓	✗	✗	✗
Repeatable Read	✓	✓	✓	✗	✗
Serializable	✓	✓	✓	✓	✓

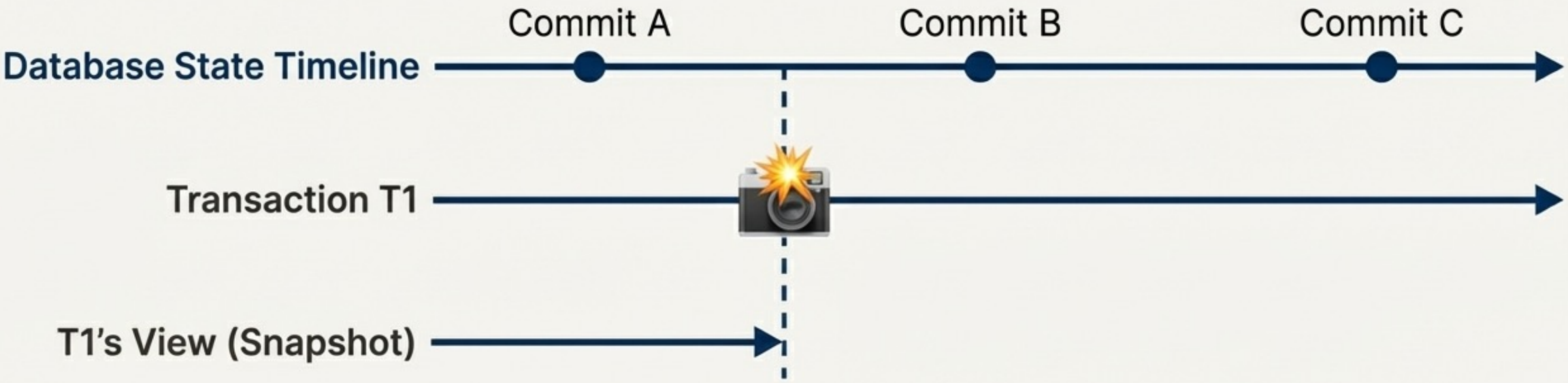
(Note: Lost updates are forbidden by the standard at all levels.)

NotebookLM

The PostgreSQL Way: Snapshot Isolation

PostgreSQL does not use the traditional locking-based model that the SQL standard was built on. It uses **Multi-Version Concurrency Control** (MVCC). Instead of locking rows, it maintains multiple versions of a row. Each transaction gets a consistent "**snapshot**" of the database at a specific point in time. It sees all data that was committed ***before*** its snapshot was taken.

A major benefit: Readers don't block writers, and writers don't block readers.
This also means **Dirty Reads are fundamentally impossible** in PostgreSQL.



NotebookLM

Level 1: Read Committed (The PostgreSQL Default)

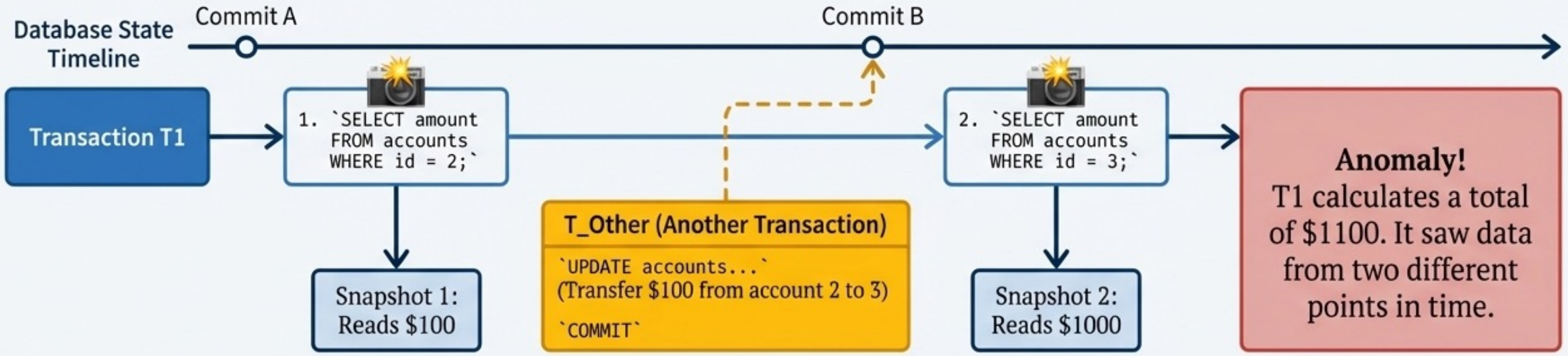
Behavior

In PostgreSQL's `Read Committed`, each **statement** gets a fresh snapshot of the database.

Guarantees

Prevents: Dirty Reads.
Allows: Non-Repeatable Reads and Phantom Reads.

The Hidden Anomaly: Read Skew



Practical Insight: In `Read Committed`, you cannot make decisions based on data you read in a previous statement. By the time your next statement runs, that data could have changed.

NotebookLM

Writing Safe Code in `Read Committed`

The classic ‘check-then-update’ anti-pattern is unsafe in `Read Committed`. There are three primary strategies to fix this.



1. Use Database Constraints

Move the logic into the database itself. Instead of application-level `IF` checks, use an `ALTER TABLE ... ADD CHECK (amount >= 0)` constraint. Let the database enforce consistency atomically.



2. Use Single, Atomic Statements

Combine the check and the update into one operation. A single statement always operates on a single snapshot. Use Common Table Expressions (CTEs) or `UPDATE ... WHERE ...` clauses.



3. Use Explicit Locking (Last Resort)

If the logic is too complex for a single statement, explicitly lock the rows you are about to modify using `SELECT ... FOR UPDATE`. This forces other transactions to wait, but sacrifices the performance benefits of MVCC.

NotebookLM

Level 2: Repeatable Read (A Consistent Snapshot)

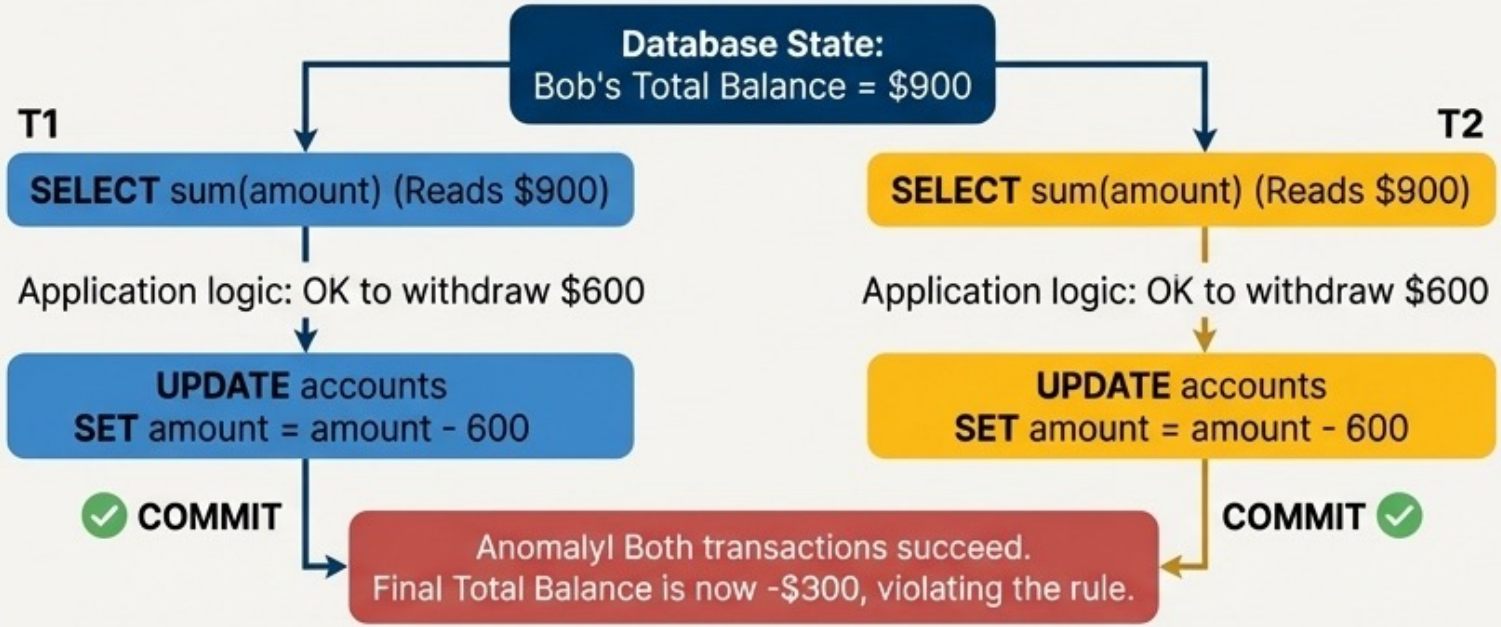
Behavior

The transaction gets one snapshot at the very beginning. Every statement sees the exact same data.

Guarantees

Prevents: Dirty Reads, Non-Repeatable Reads, and even Phantom Reads. (PostgreSQL's implementation is stricter than the standard here).

The Hidden Anomaly: Write Skew



The Trade-Off: Serialization Failures



If T1 and T2 had tried to write to the *same* row, one would be aborted with a `serialization failure` error.

```
ERROR: could not serialize access
due to concurrent update
```

Practical Insight: If your application uses `Repeatable Read` for write transactions, it *must* include logic to catch serialization failures and retry the entire transaction.

NotebookLM

Level 3: Serializable (The Fortress of Consistency)

Behaves like `Repeatable Read` (one snapshot per transaction) but adds an advanced mechanism to detect and prevent anomalies like Write Skew.

Guarantees

Prevents: All known and even unknown concurrency anomalies.

The result is guaranteed to be the same as if the transactions had run one after another in some order.



How It Works

The database tracks dependencies between transactions.

If it detects a situation that *could* lead to an anomaly, it will pre-emptively abort one of the transactions with a `serialization failure`.

The Cost

Performance Overhead: This dependency tracking adds significant overhead.

More Retries: Failures are more frequent than in `Repeatable Read`, requiring a robust retry framework.

Limitations: `Serializable` transactions cannot be used on read-only replicas.

NotebookLM

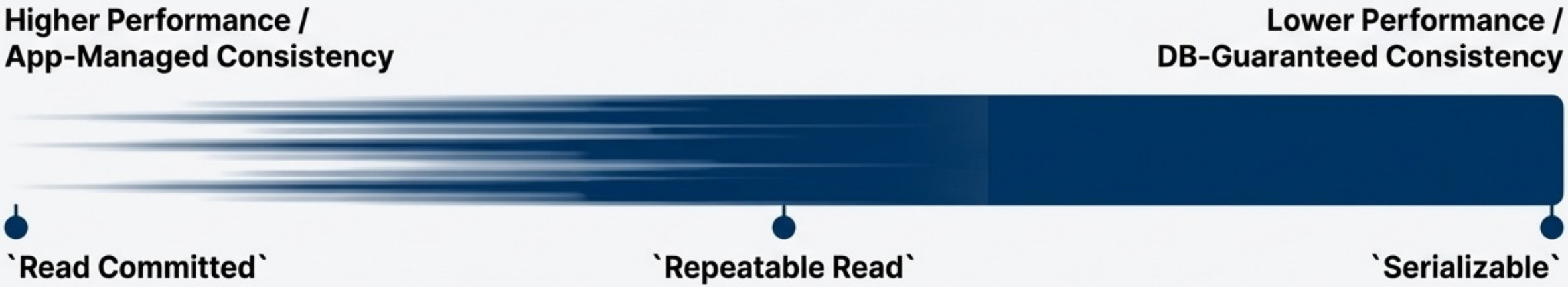
Summary: Isolation Levels in PostgreSQL

Level	Snapshot Behavior	Key Anomalies Allowed / Side Effects	Main Drawback
Read Committed	New snapshot **per statement**	Non-Repeatable Reads, Phantom Reads. Prone to **Read Skew**.	Application code must be written very carefully to ensure consistency.
Repeatable Read	One snapshot **per transaction**	Prevents standard anomalies. Prone to **Write Skew**.	Transactions can fail with **serialization failures** and must be retried.
Serializable	One snapshot **per transaction**	Prevents **all anomalies**, including Write Skew.	More frequent **serialization failures** and higher performance overhead.

NotebookLM

The Developer's Dilemma: Which Level Should You Use?

The choice is a fundamental trade-off between performance and the complexity of ensuring data consistency.



Use `Read Committed` (Default) when...

Performance is critical and you're confident your application logic can prevent anomalies through careful coding. You accept responsibility for consistency.

Use `Repeatable Read` when...

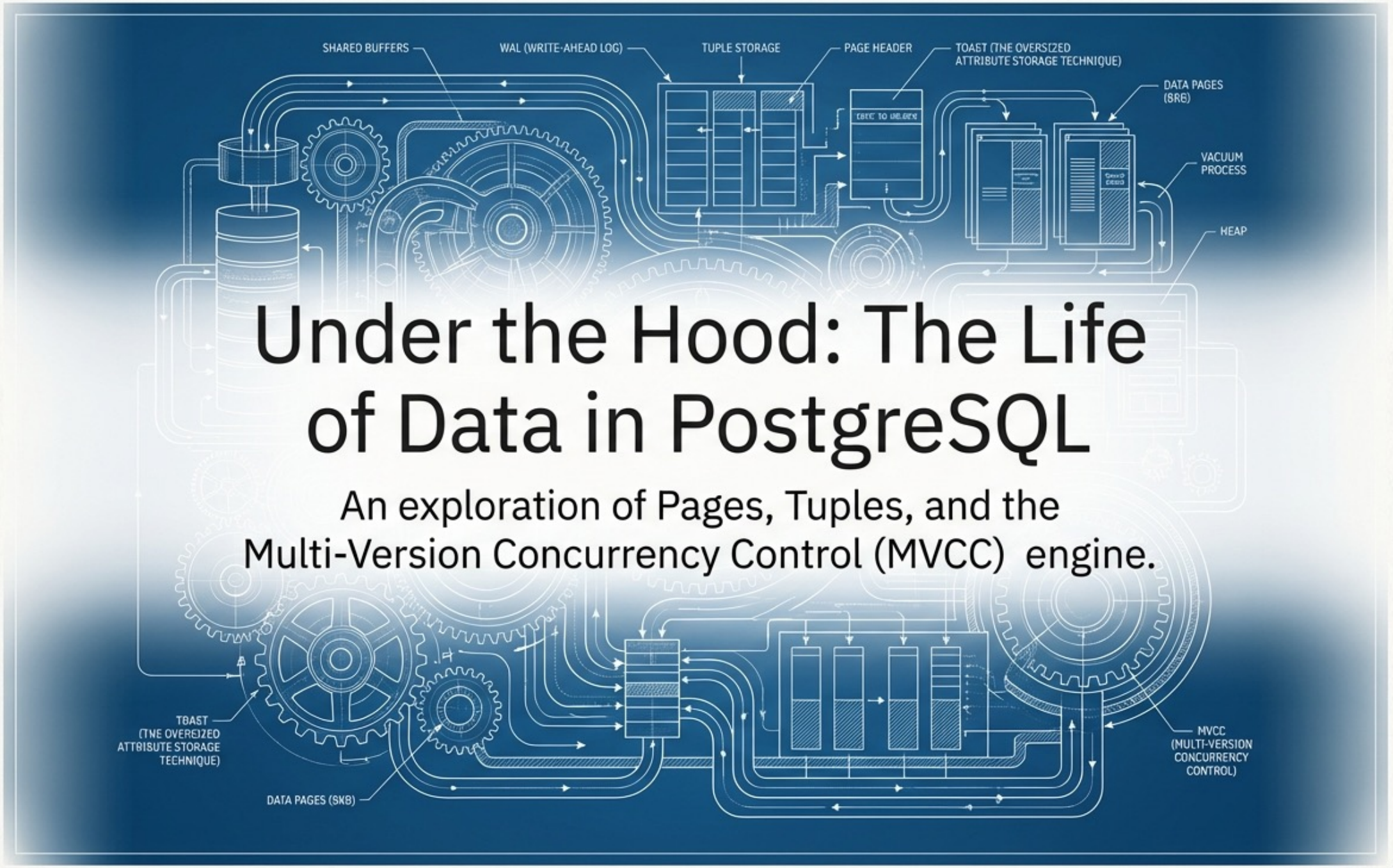
You have long-running, read-only transactions (like reports) that need a stable view of data. For writes, use only if you have a solid transaction retry mechanism.

Use `Serializable` when...

Data consistency is absolutely paramount and worth the performance cost and engineering effort of a robust, automated retry framework.

NotebookLM

3. Pages and Tuples

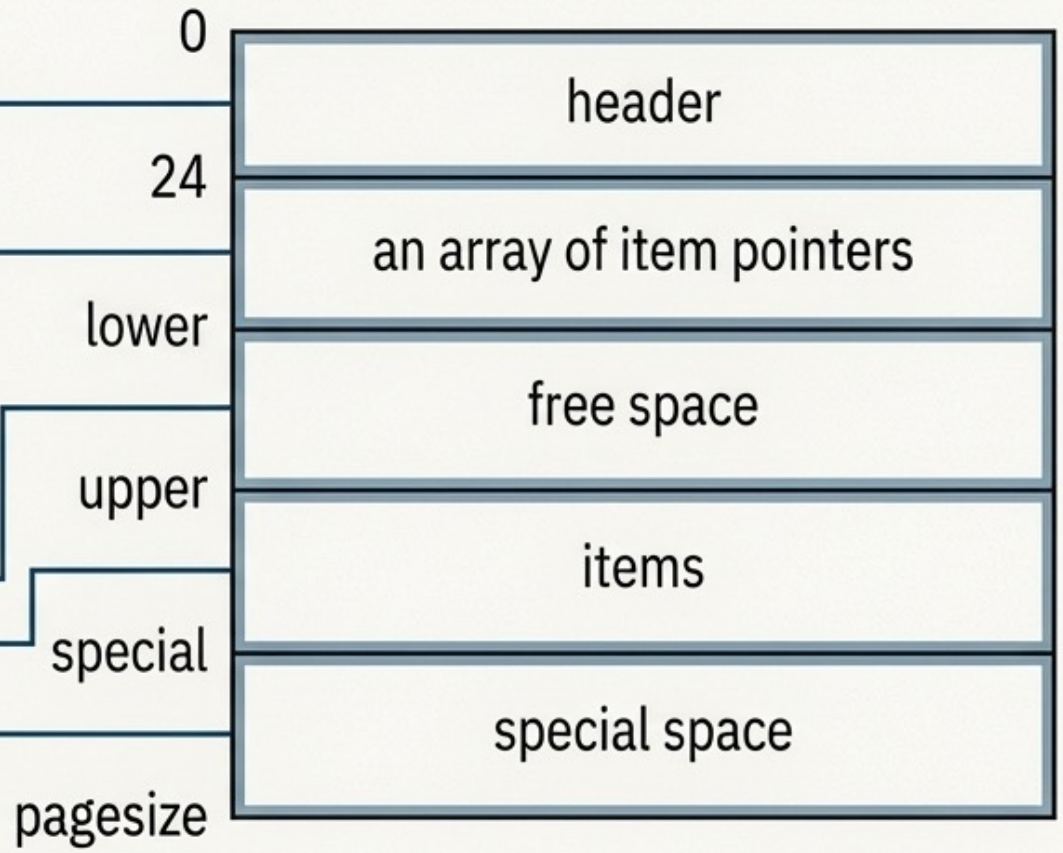

SHARED BUFFERS
WAL (WRITE-AHEAD LOG)
TUPLE STORAGE
PAGE HEADER
TOAST (THE OVERSIZED ATTRIBUTE STORAGE TECHNIQUE)
DATA PAGES (8KB)
VACUUM PROCESS
HEAP
Under the Hood: The Life of Data in PostgreSQL
An exploration of Pages, Tuples, and the Multi-Version Concurrency Control (MVCC) engine.
TOAST (THE OVERSIZED ATTRIBUTE STORAGE TECHNIQUE)
DATA PAGES (8KB)
MVCC (MULTI-VERSION CONCURRENCY CONTROL)

NotebookLM

The Blueprint: Everything Lives on a Page

PostgreSQL organizes all data—from tables to indexes—into fixed-size blocks, typically 8KB, called Pages. This is the fundamental unit of I/O. The structure of a page is meticulously organized to manage data efficiently.

- **Page Header:** Metadata about the page.
- **Item Pointer Array:** An indirection layer for locating data.
- **Free Space:** A contiguous block for new data.
- **Items (Tuples):** The actual row data, added from the end of the page.
- **Special Space:** Reserved for specific index access methods.

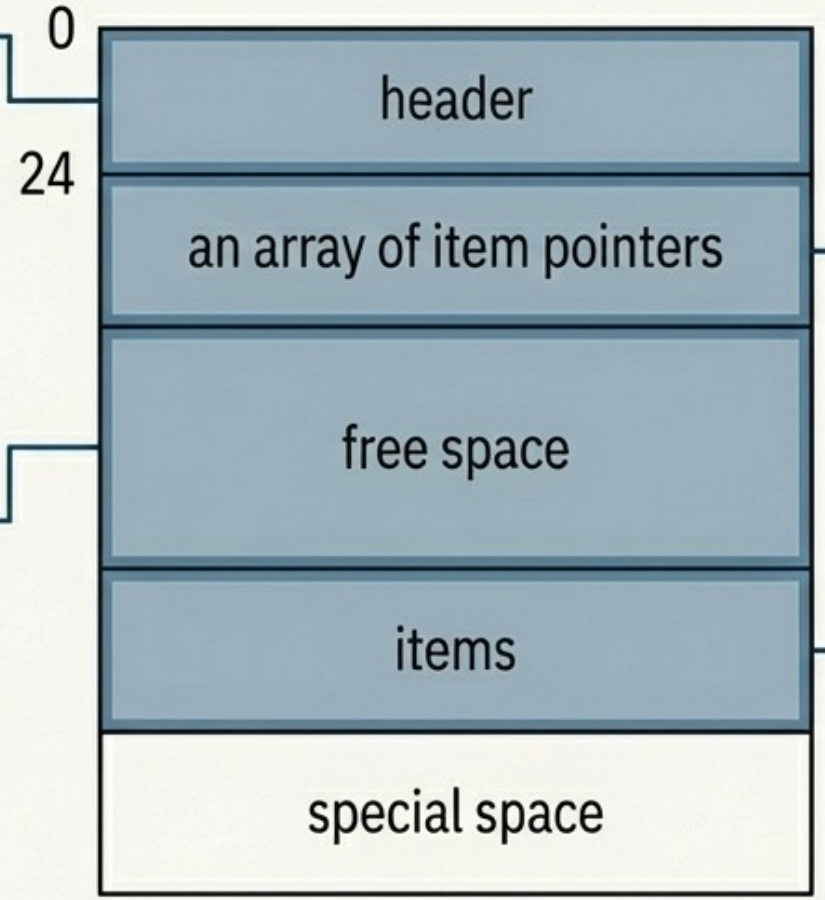


NotebookLM

Anatomy of a Page: A Guided Tour

1. Header
24 bytes of critical metadata, including checksums and pointers (`lower, `upper`) that track the boundaries of the used space.

4. Free Space
The contiguous area between the last item pointer and the first tuple. PostgreSQL ensures this is always one continuous block.



2. Item Pointers (4 bytes each)
This is the genius of the design. Instead of pointing directly to a row, indexes use a Tuple Identifier (TID) composed of (page #, item pointer #).

The 'Why': This indirection allows PostgreSQL to move a tuple *within* the page to compact free space (defragment), only needing to update the small 4-byte pointer. The external TID reference remains stable.

3. Items (Tuples / Row Versions)
The actual data, which grows 'backwards' from the end of the page.

NotebookLM

Inside a Tuple: More Than Just Your Data

What we call a “row” is stored as a “tuple” or, more accurately, a “row version.” This distinction is central to PostgreSQL’s MVCC. Each tuple carries a significant metadata header.



xmin: The Transaction ID (XID) that created this row version.

xmax: The XID that deleted this row version. (0 if not deleted).

ctid: A pointer to the next version of this same row (used in `UPDATE`s).

infomask: A bitmask of flags tracking transaction status (committed, aborted) and other states.

Null Bitmap: Indicates which columns are NULL.

Key Takeaway

This header requires a minimum of 23 bytes per tuple. For tables with few or narrow columns, this overhead can easily exceed the size of the actual user data.

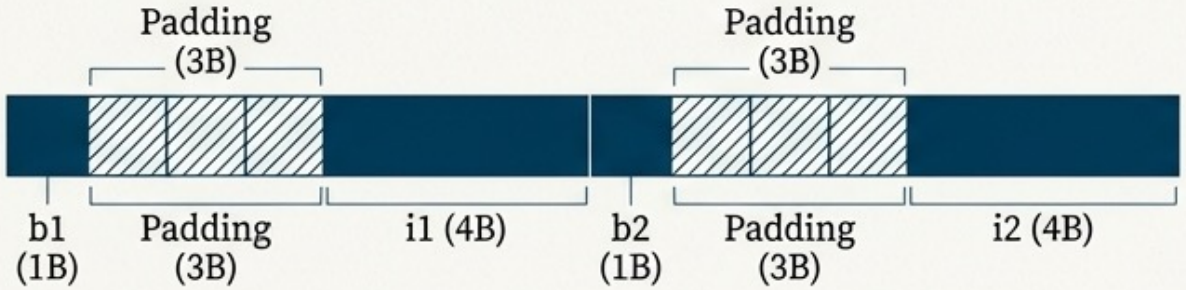
NotebookLM

The Hidden Cost of Misalignment

The physical size of a tuple depends on data alignment to machine-word boundaries.
The order of columns in your `CREATE TABLE` statement matters.

Inefficient Ordering

```
CREATE TABLE padding (
  b1 boolean, i1 integer,
  b2 boolean, i2 integer
);
INSERT INTO padding VALUES (true,1,false,2);
```

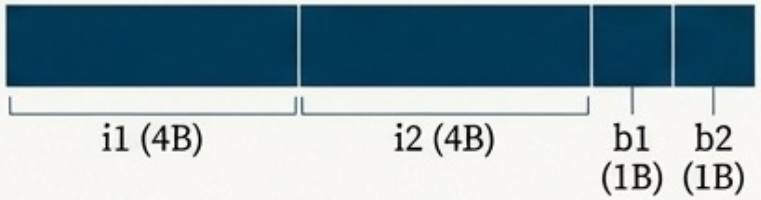


Result: 40 bytes

6 bytes are wasted as padding to align the integer columns.
(Header: 24B, Integers: 2*4B, Booleans: 2*1B = 34B. 40-34=6B wasted).

Efficient Ordering

```
CREATE TABLE padding (
  i1 integer, i2 integer,
  b1 boolean, b2 boolean
);
```



Result: 34 bytes

Conclusion: Grouping fixed-size columns (especially larger ones) first can reduce padding and save significant space.

NotebookLM

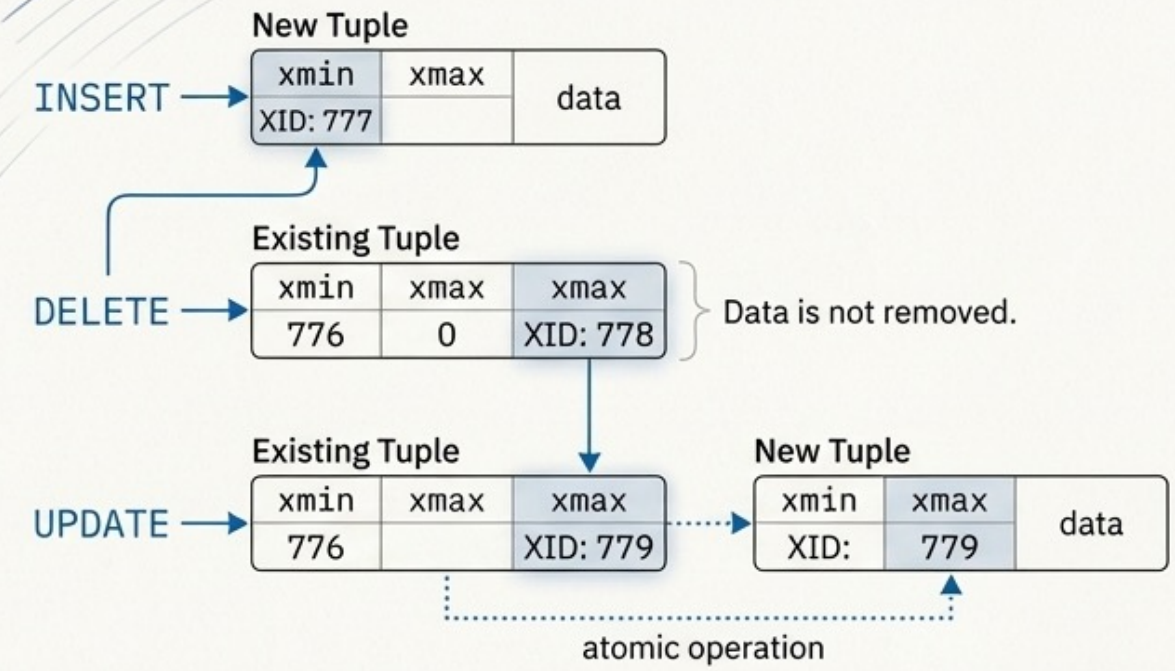
The Engine in Motion: Transactions Define Reality

The static structure of pages and tuples comes to life through transactions. PostgreSQL uses Multi-Version Concurrency Control (MVCC) to ensure readers don't block writers, and writers don't block readers.

The Golden Rules of Tuple Visibility

- Visibility is **not** determined by time, but by **Transaction IDs** (XIDs).
- `xmin` stores the XID of the transaction that created the tuple.
- `xmax` stores the XID of the transaction that deleted the tuple.

How Operations Translate



NotebookLM

A Tuple is Born: The `INSERT`

Scenario

```
BEGIN;
INSERT INTO t(s) VALUES ('FOO');
-- Current Transaction ID is 776
```

Microscopic View

We use a helper function wrapping pageinspect to view the tuple header.

```
-- Helper Function for Readability
CREATE FUNCTION heap_page(relname text, pageno integer) ...
```

Output & Analysis

```
SELECT * FROM heap_page('t', 0);

 ctid  | state  | xmin | xmax
-------+--------+------+-------
 (0,1) | normal |  776 |    0 a
```

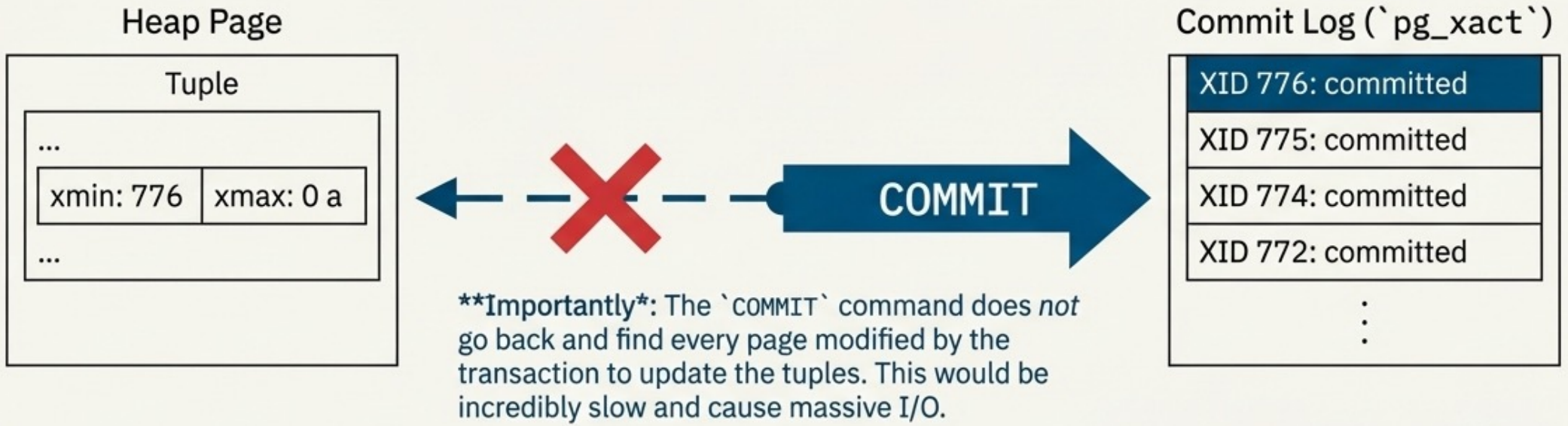
`xmin` is `776`: The tuple was created by our current, uncommitted transaction.

`xmax` is `0 a`: The 'a' indicates `xmax_aborted`. This is a placeholder; the tuple has not been deleted.

Crucially: There is no 'c' (committed) next to `xmin`. The hint bits are not set yet because the transaction's outcome is unknown.

NotebookLM

Making it Permanent: `COMMIT` and the `pg_xact` Log



**Importantly*: The `COMMIT` command does *not* go back and find every page modified by the transaction to update the tuples. This would be incredibly slow and cause massive I/O.

State of the Page After `COMMIT`

```
COMMIT;
SELECT * FROM heap_page('t', 0);
 ctid  | state  | xmin | xmax
-------+--------+------+------
 (0,1) | normal |  776 |  0 a
```

The page is unchanged! The `xmin` field is still just `776`. The database knows XID 776 is committed, but the tuple on the page doesn't reflect that yet.

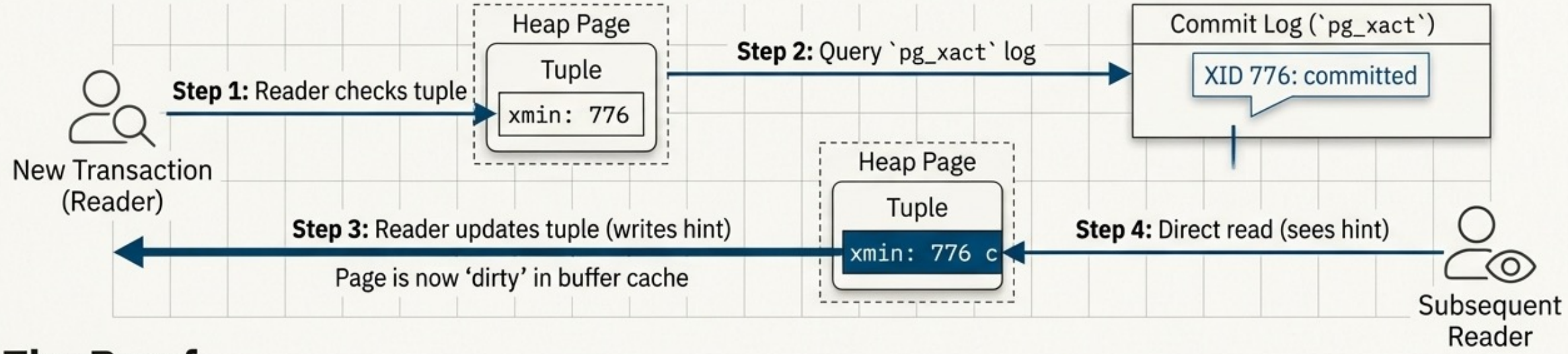
NotebookLM

The Reader Pays the Bill: How Hint Bits Get Set

The Problem

How does the next transaction know if XID 776 is committed? Without a hint, it would have to check the `pg_xact` log every single time, which is too slow.

The Solution: Hint Bits



The Proof

```
-- A simple SELECT triggers the hint bit update
SELECT * FROM t;

-- Now the page reflects the committed state
SELECT * FROM heap_page('t', 0);
```

```
 ctid  | state  | xmin  | xmax
-------+--------+-------+------
 (0,1) | normal | 776 c | 0 a
```

NotebookLM

The `UPDATE`: A Tale of Two Tuples

An `UPDATE` is atomically a `DELETE` of the old version and an `INSERT` of the new version, both within the same transaction.

Scenario

```
BEGIN; -- New Transaction ID is 778
UPDATE t SET s = 'BAR';
```

View from the Page

```
SELECT * FROM heap_page('t', 0);

 ctid  | state  | xmin  | xmax
-------+--------+-------+-------
 (0,1) | normal | 776 c |  778
 (0,2) | normal | 778   |    0 a
```

Analysis

The original tuple. Its `xmax` is now set to `778`, marking it as deleted by our current transaction. This also acts as a row-level lock.

The new version of the row. Its `xmin` is `778`, created by our current transaction.

Other transactions running with a snapshot taken *before* XID 778 will still see tuple `(0,1)`.

NotebookLM

The Ghost in the Machine: `DELETE` and `ROLLBACK`

Scenario

```
BEGIN; -- New Transaction ID is 777
DELETE FROM t;

-- At this point, the page looks like this:
-- ctid | state  | xmin  | xmax
-- -----+--------+-------+------
-- (0,1)| normal | 776 c | 777

ROLLBACK;
```

Final State

```
-- A SELECT triggers the hint bit update
SELECT * FROM t;

SELECT * FROM heap_page('t', 0);
 ctid  | state  |  xmin  |  xmax
-------+--------+--------+--------
 (0,1) | normal | 776 c  | 777 a
```

The Aftermath



The next transaction that reads the tuple will check `xmax` (`777`), find it was aborted, and set the `xmax_aborted` hint bit.

Conclusion

The tuple is now fully visible again. The change from the aborted transaction remains on disk but is ignored by all future queries.

NotebookLM

The Ripple Effect: How Indexes See a Multi-Version World

Indexes do **not** use `xmin`/`xmax` or MVCC internally. An index entry is a simple pointer (TID) to a specific version of a row on a heap page.

Key Behavior

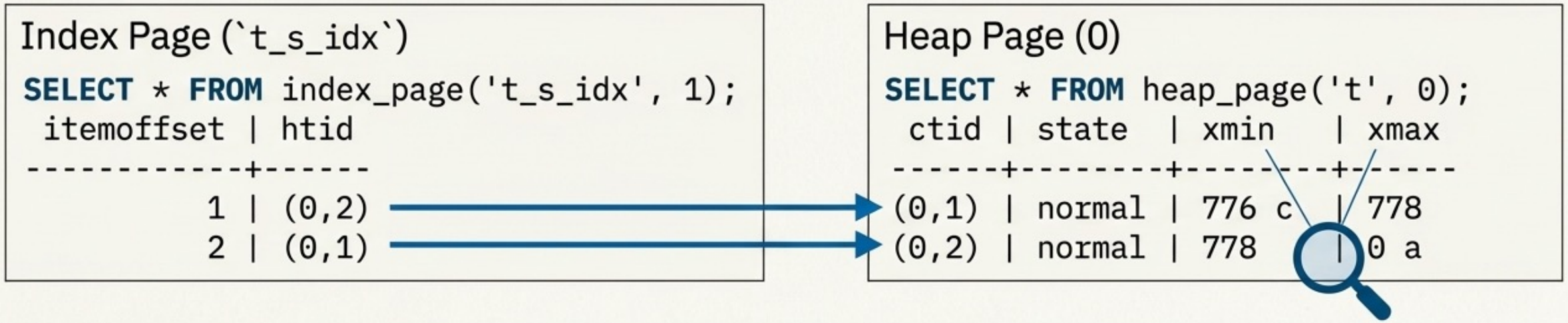
When a row is updated, a new entry is added to the index for the new tuple version. The old one is not immediately removed.

Index Page (`t_s_idx`)

```
SELECT * FROM index_page('t_s_idx', 1);
 itemoffset | htid
------------+------
          1 | (0,2)
          2 | (0,1)
```

Heap Page (0)

```
SELECT * FROM heap_page('t', 0);
 ctid | state  | xmin    | xmax
------+--------+---------+------
(0,1) | normal | 776 c   | 778
(0,2) | normal | 778     | 0 a
```



The Process

A query using the index will fetch the TIDs, visit the heap page, and then perform the visibility check (`xmin`, `xmax`, hint bits) to determine which version (if any) it can see.

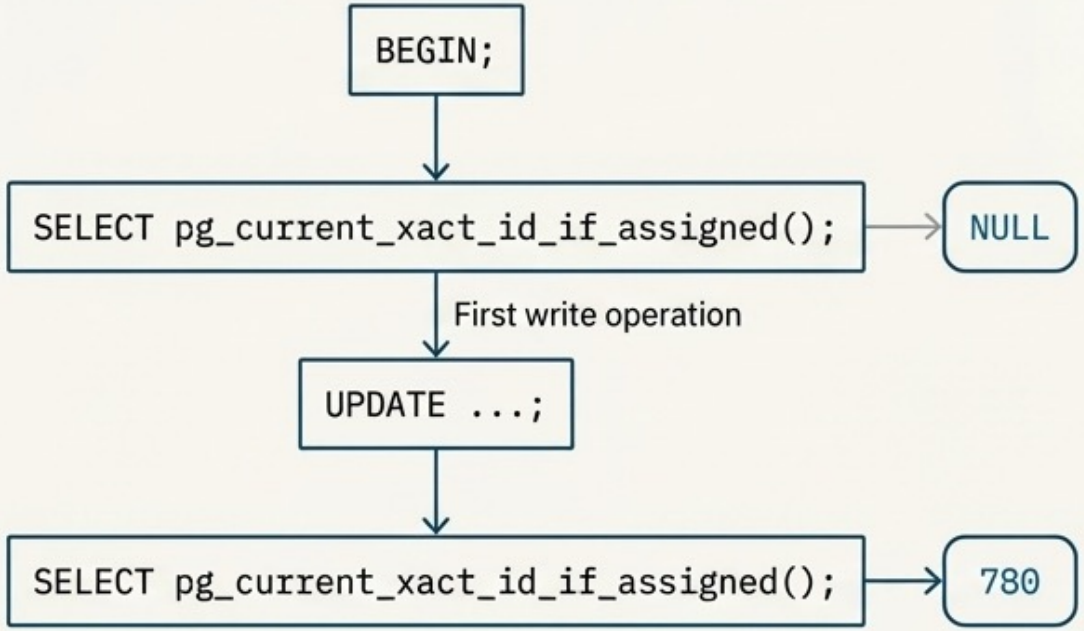
NotebookLM

Beyond the Basics: Specialized Transactions

Virtual Transactions (An XID-Saving Trick)

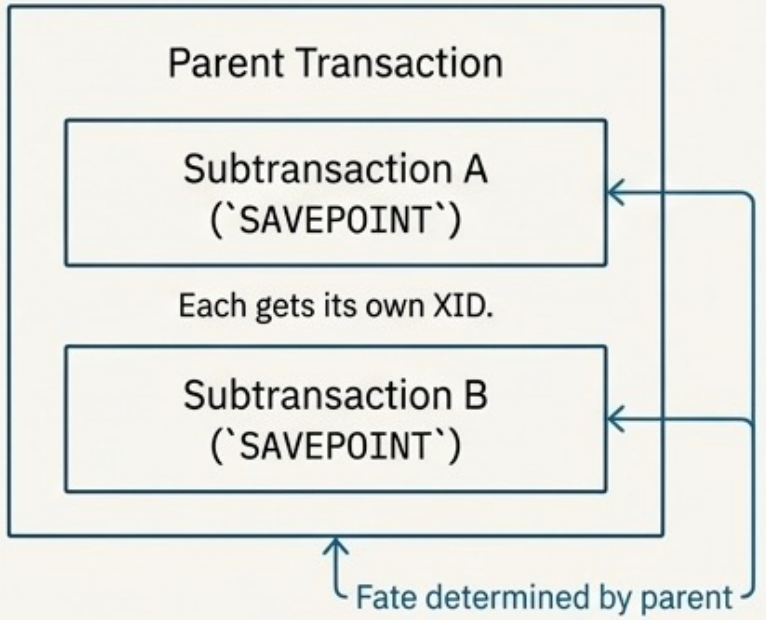
A read-only transaction doesn't initially consume a permanent Transaction ID. It is assigned a 'virtual' XID, local to its backend process.

Assigning a real XID requires a system-wide lock. Avoiding this for read queries is a major performance gain and reduces XID wraparound pressure.



Subtransactions (`SAVEPOINT`)

`SAVEPOINT`s are implemented as child transactions, each getting its own XID.



The ultimate fate of a committed subtransaction depends on its parent. If the parent transaction aborts, all its subtransactions are considered aborted, regardless of their individual state.

NotebookLM

Core Principles of PostgreSQL Storage



1. Pages are the Foundation

What: All data is stored in fixed-size pages.

Why: Aligns I/O with the operating system and simplifies buffer management.



2. MVCC Creates Parallel Universes

What: UPDATE is a DELETE+INSERT, creating new row versions instead of overwriting.

Why: Guarantees that readers never block writers, enabling high concurrency.



3. Indirection Provides Stability

What: Item Pointers (TIDs) decouple a row's logical identifier from its physical location on a page.

Why: Allows for on-page defragmentation without needing to update countless index entries.



4. Lazy Updates Win the Race

What: Hint bits are set by the next reader, not by the committing transaction.

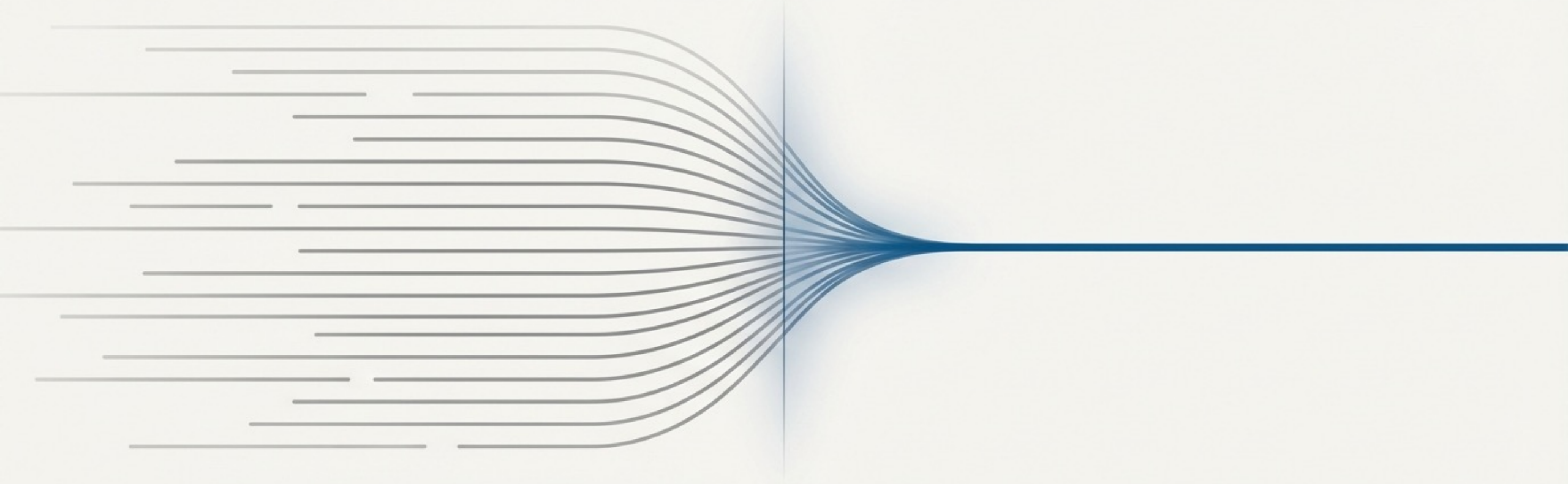
Why: Keeps the COMMIT operation extremely fast by deferring page-write overhead from its critical path.

NotebookLM

4. Snapshots

Decoding PostgreSQL Snapshots

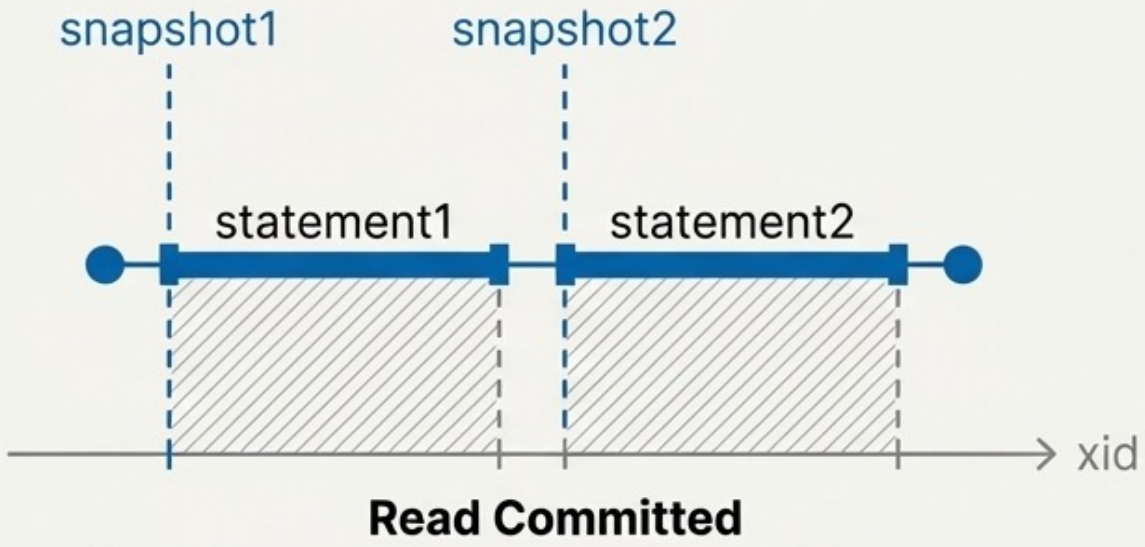
How the database provides a consistent view of your data without looking at the clock.



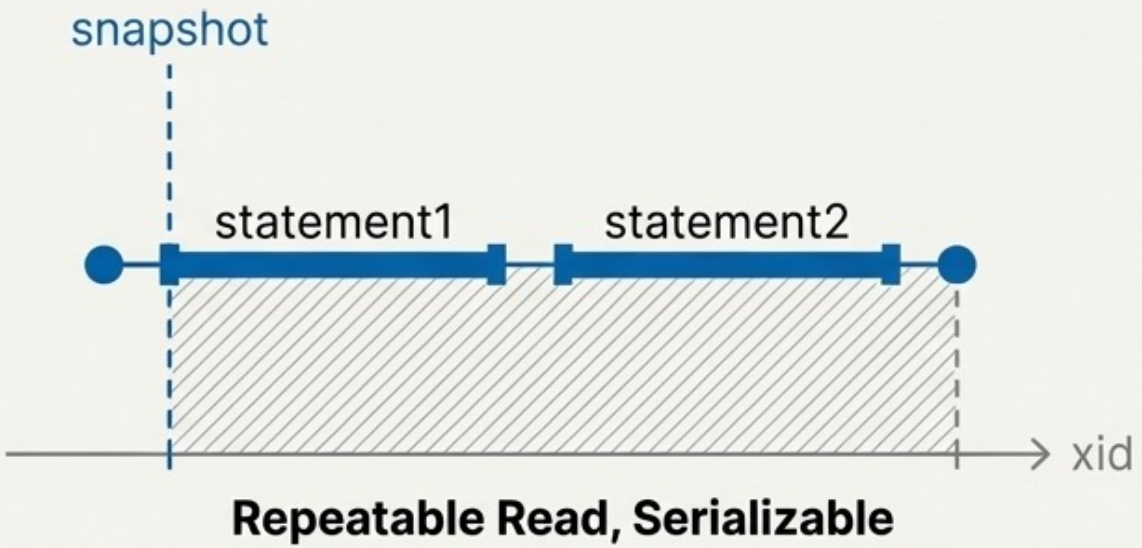
NotebookLM

A Snapshot's Lifespan Depends on Your Isolation Level

A transaction doesn't just see "the latest data." It sees data from a specific snapshot, and the **timing** of that snapshot is dictated by the isolation level.



Read Committed



Repeatable Read, Serializable

Read Committed

A new snapshot is taken at the beginning of *every statement*. The snapshot lasts only for the duration of that single statement.

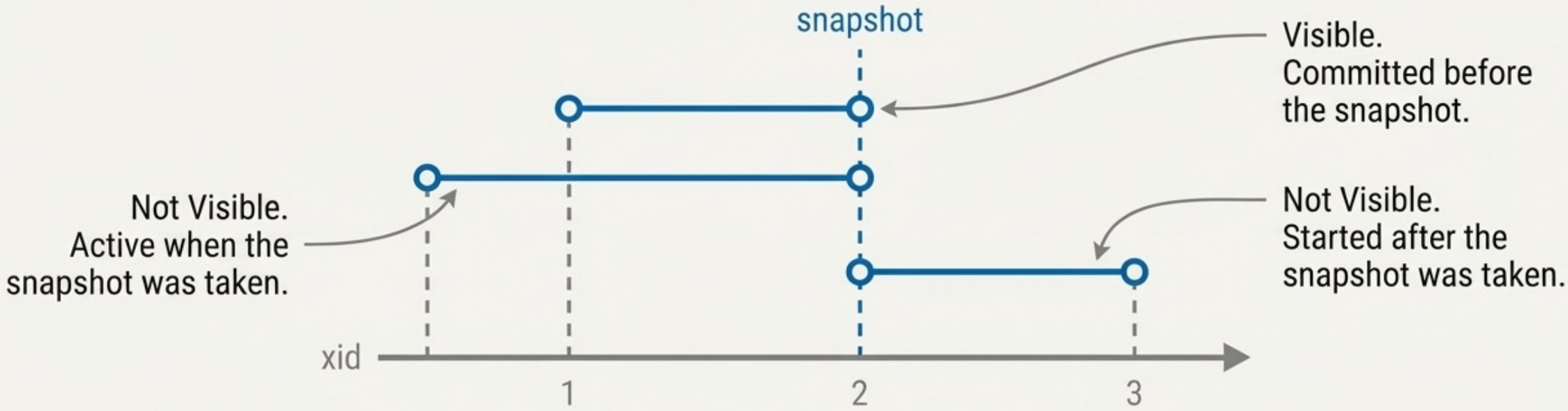
Repeatable Read / Serializable

A single snapshot is taken at the start of the *first statement* in the transaction. This one snapshot is used for the entire duration of the transaction.

NotebookLM

The Intuitive Model: Visibility Based on Time

It seems logical that a snapshot's visibility rules are based on time. A transaction's changes are visible if it committed *before* the snapshot was taken.



This simple timeline seems to explain everything. But it's not how PostgreSQL works.

NotebookLM

The Reality: PostgreSQL Doesn't Use Commit Time for Visibility

The system fundamentally does not know *when* a transaction was committed. While a transaction's start time is known (its ID), its completion time is not recorded in a way that is used for visibility checks.

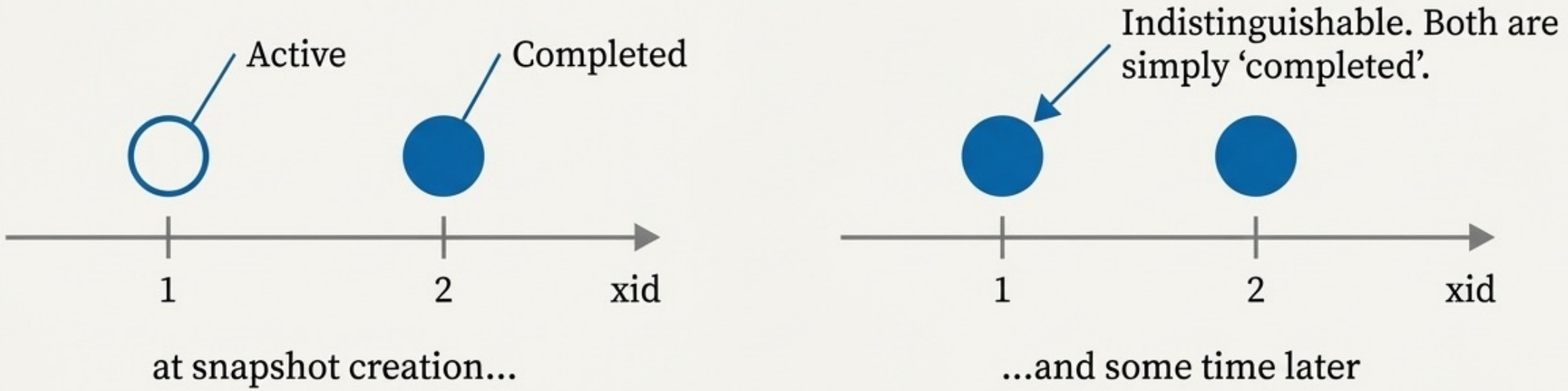
- The `track_commit_timestamp` parameter exists but is not used for visibility checks. It is primarily for external tools like replication solutions.
- Commit times are recorded in the WAL (Write-Ahead Log), but this is for Point-In-Time Recovery, not for live transaction visibility.

> "Unfortunately, the previous picture has nothing to do with how PostgreSQL actually sees it."

NotebookLM

The Problem of The Problem of Lost State

All the system knows is the state of transactions *right now*. This information is held in a shared memory structure (ProcArray). Once a transaction completes, the fact that it *was* active at a specific past moment is lost.



This is why PostgreSQL cannot perform "flashback" or retrospective queries to see the database state at an arbitrary past time, even if the old row versions exist on disk.

NotebookLM

The Real Anatomy of a Snapshot

Instead of a single point in time, a snapshot is a set of rules defined by three key values, captured at the moment of its creation.



`xmin`

The transaction ID of the oldest active transaction at the time of the snapshot.

- Any transaction with an ID less than `xmin` is considered “in the past” and is either committed and visible, or aborted and invisible.



`xmax`

The next available transaction ID. It’s one greater than the latest completed transaction ID.

- Any transaction with an ID greater than or equal to `xmax` is “in the future” and is not visible.



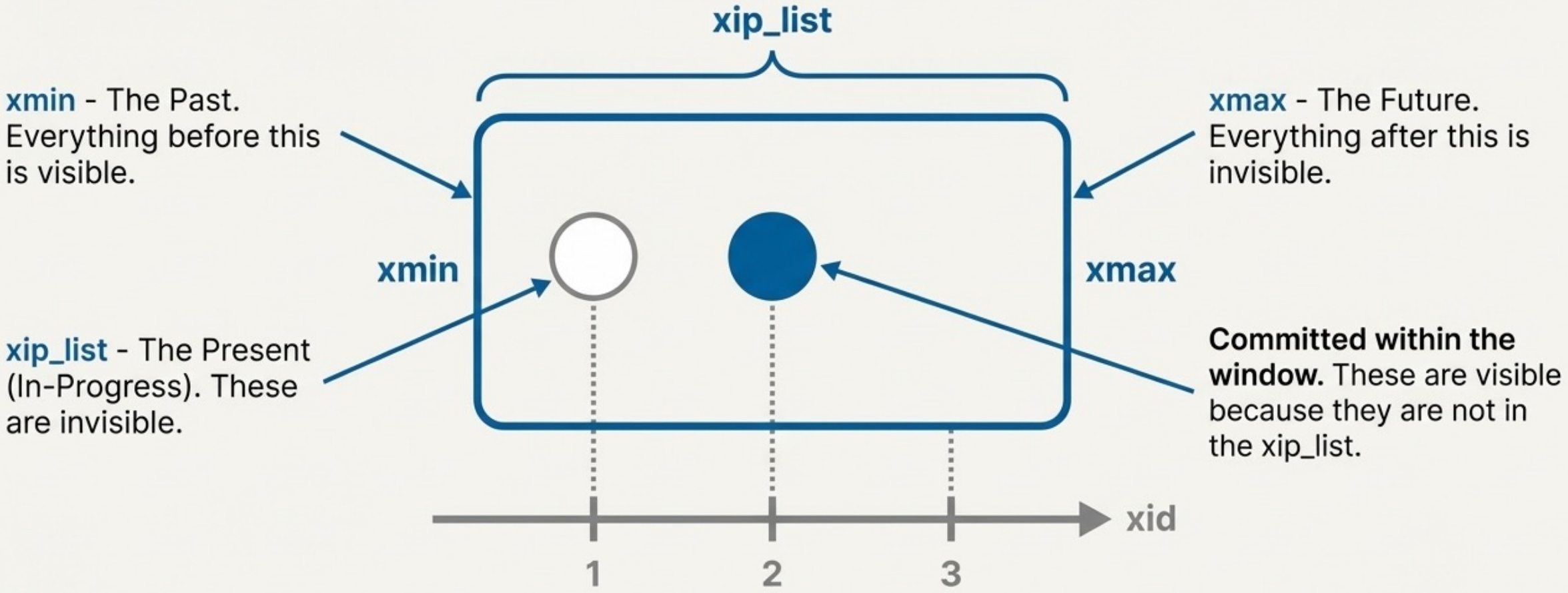
`xip_list`

A list of transaction IDs that were active at the time of the snapshot and fall between `xmin` and `xmax`.

- These are “in-progress” transactions, and their changes are not visible.

NotebookLM

Visualizing the Snapshot's Rules



A tuple is visible to this snapshot if its creating transaction is committed, is not in the xip_list, and has an xid greater than or equal to *xmin* and less than *xmax*.

NotebookLM

A Practical Example: Capturing a Snapshot

Scenario Setup

1. **Transaction 790 (Session 1):**
   `BEGIN;`
   `INSERT INTO accounts ...;`
   (Remains open).
2. **Transaction 791 (Session 2):**
   `BEGIN;`
   `INSERT INTO accounts ...;`
   `COMMIT;`
   (Completes).

Capturing the Snapshot

A new transaction begins in Session 3 at the `REPEATABLE READ` isolation level, capturing the snapshot.

```
=> SELECT pg_current_snapshot();
 pg_current_snapshot
---------------------
         790:792:790
```

- `xmin`: `790` — The oldest active transaction is 790.
- `xmax`: `792` — The next available transaction ID is 792.
- `xip_list`: `790` — Transaction 790 is currently in progress.

NotebookLM

Applying the Rules: What the Snapshot Sees

The Query Result

A `SELECT * FROM accounts` using our snapshot (790:792:790) sees only one row.

```
=> SELECT ctid, * FROM accounts;
 ctid  | id | client | amount
-------+----+--------+--------
 (0,2) |  2 | bob    | 100.00
```

The Ground Truth

The table physically contains three row versions (tuples).

- **(0,1):** created by `xmin` 790
- **(0,2):** created by `xmin` 791, deleted by `xmax` 792
- **(0,3):** created by `xmin` 792

Why the Discrepancy?

- **Tuple (0,1) / xid 790**: Invisible. Its `xmin` (790) is within bounds, but it's in the `xip_list`.
- **Tuple (0,2) / xid 791**: Visible. Its `xmin` (791) is within bounds and not in the `xip_list`. Its deletion by `xmax` 792 is not visible because **792** ≥ our snapshot's `xmax`.
- **Tuple (0,3) / xid 792**: Invisible. Its `xmin` (792) is ≥ our snapshot's `xmax`.

NotebookLM

The Transaction Horizon

Concept

The `xmin` of the oldest active snapshot in the *entire database* defines a critical boundary called the “database horizon”. This concept is inspired by the “event horizon” in physics.

What it Means

Any tuple version created by a transaction older than this horizon is no longer visible to *any* possible transaction currently running in the database.

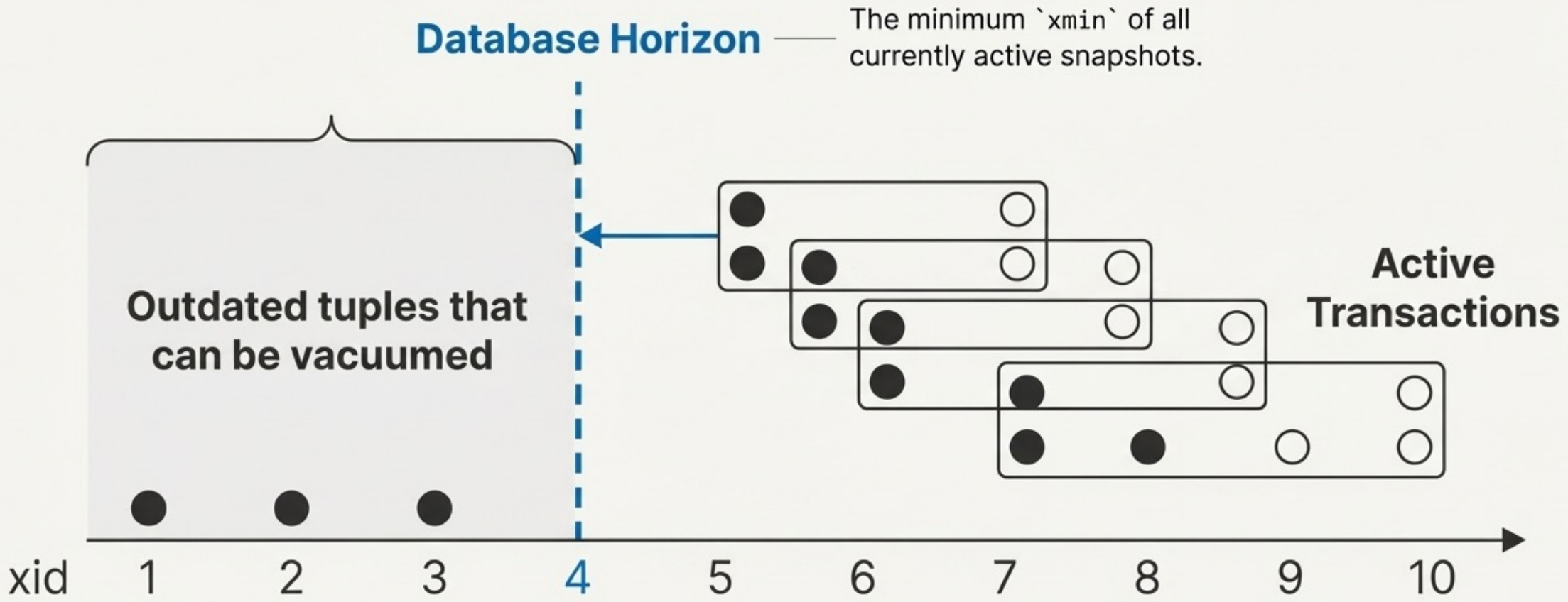
The Critical Implication

These old, invisible tuples are now ‘dead’. They are candidates for removal by the `VACUUM` process.

The progress of `VACUUM` is directly tethered to the position of this horizon.

NotebookLM

How the Horizon Enables VACUUM



The `VACUUM` process can only clean up dead tuples that fall behind the database horizon. If the horizon is held back by a long-running transaction, cleanup cannot proceed, leading to table and index bloat.

NotebookLM

What Holds the Horizon Back?

Anything that **maintains an old snapshot prevents the database horizon** from **advancing,** which in turn blocks `VACUUM`.



1. **Long-running analytical queries**
   A single SELECT that runs for hours in a REPEATABLE READ or SERIALIZABLE transaction will hold a very old xmin.



2. **"Idle in transaction" sessions**
   A client that begins a transaction (BEGIN;`), performs a query, and then waits for user input without committing or rolling back. This is a common application-level bug.



3. **Prepared transactions**
   Transactions left in a prepared but uncommitted state.

A single misbehaving session can impact the entire database's health by preventing cleanup across all tables.

NotebookLM

Advanced Use Case: Synchronizing Transactions with a Shared Snapshot

The Problem

How do you ensure multiple parallel processes see the exact same consistent view of the database, even if they start milliseconds apart?

The Mechanism

PostgreSQL allows a transaction to 'export' its snapshot, generating an identifier string. Other transactions can then 'import' this snapshot.

Key Functions

pg_export_snapshot()

Returns the ID of the current transaction's snapshot.

```
SELECT pg_export_snapshot();
```

SET TRANSACTION SNAPSHOT '...'

Must be the first command in a new `REPEATABLE READ` or `SERIALIZABLE` transaction. Forces the transaction to use the

```
SET TRANSACTION SNAPSHOT '00000004-0000006E-1';
```

Primary Application

Used by utilities like `pg_dump` in parallel mode to guarantee a consistent backup across multiple worker processes.

NotebookLM

A Special Case: System Catalog Snapshots

The Rule
Regular transaction snapshots are NOT used for accessing system catalogs (like `pg_class`, `pg_attribute`).

The Reason
A transaction must always see the most up-to-date schema. If it used its own older snapshot, it might miss a new `NOT NULL` constraint or see an old table definition, leading to data corruption or incorrect behavior.

Example
1. `BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;` (Snapshot is taken)
2. Another session runs `ALTER TABLE accounts ADD CONSTRAINT ...;`
3. The first transaction now runs an `INSERT` that violates the new constraint.

****Result****
The `INSERT` will correctly fail. The transaction 'sees' the new constraint, even though it was created after its snapshot was taken.

Behavior
In effect, each catalog query uses its own, up-to-the-moment snapshot to ensure consistency.

NotebookLM

The Cardinal Rule: Your Transaction's Lifetime Dictates Database Health

- **Summary of the Story**
  We've seen that PostgreSQL's MVCC doesn't rely on time, but on a sophisticated system of transaction state (`xmin`, `xmax`, `xip_list`).
- **The Core Consequence**
  This design directly links the lifetime of your longest-running transaction to the efficiency of your database's storage management.

Actionable Takeaway
Proactively monitor for and eliminate long-running and 'idle in transaction' states. This isn't just about reducing locks; it's the single most important practice to prevent table and index bloat and ensure `VACUUM` can do its job effectively.

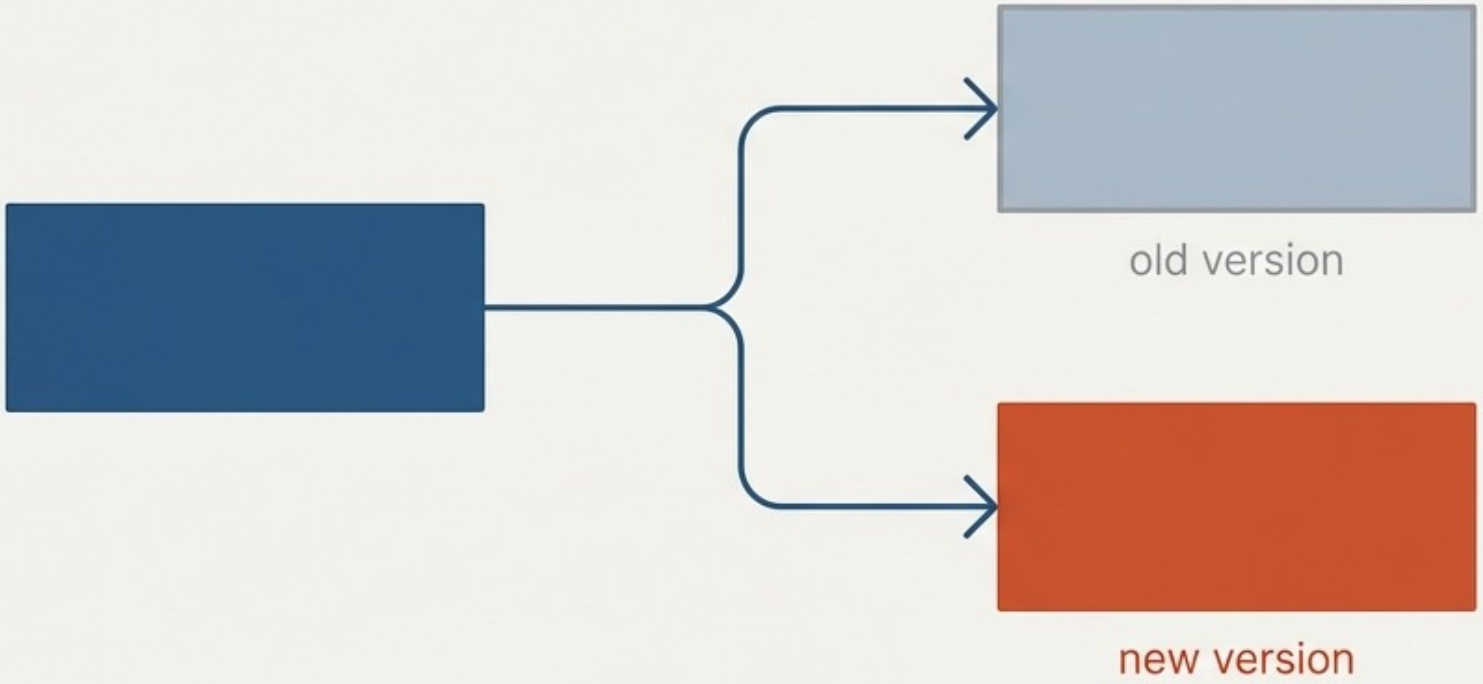
A healthy PostgreSQL database is one with short-lived transactions.

NotebookLM

5. Page Pruning and HOT Updates

Understanding PostgreSQL's Hidden Update Machinery

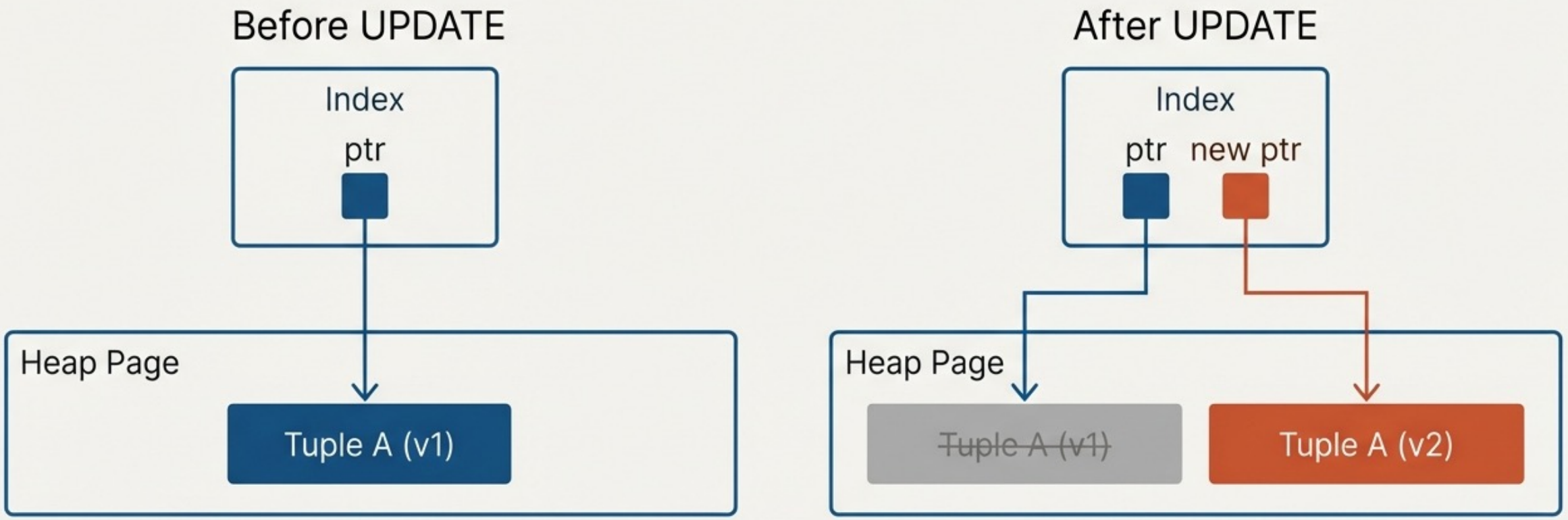
A Deep Dive into Page Pruning and Heap-Only Tuple (HOT) Updates



NotebookLM

Every UPDATE is a DELETE + INSERT

PostgreSQL never modifies data in place. To preserve transaction isolation (MVCC), an UPDATE operation marks the old version of a row as "dead" (or expired) and inserts a new version.



NotebookLM

This Process Creates Dead Tuples, Leading to Bloat

As a row is updated repeatedly, old versions accumulate within the data page. These “dead tuples” consume space but are no longer visible to new transactions.

```
CREATE TABLE hot(id integer, s char(2000));
CREATE INDEX hot_id ON hot(id);
CREATE INDEX hot_s ON hot(s);

INSERT INTO hot VALUES (1, 'A');
UPDATE hot SET s = 'B';
UPDATE hot SET s = 'C';
UPDATE hot SET s = 'D';
```

```
=> SELECT * FROM heap_page('hot',0);
 ctid  | state  |  xmin |  xmax
-------+--------+-------+-------
 (0,1) | normal | 801 c | 802 c
 (0,2) | normal | 802 c | 803 c
 (0,3) | normal | 803 c | 804
 (0,4) | normal | 804   |   0 a
```

Three dead versions of the row still occupy space. Only (0,4) is live.

NotebookLM

How does Postgres reclaim this space without a full `VACUUM`?

PostgreSQL can perform a rapid ‘Page Pruning’ during any read or write operation on a page. This micro-cleanup occurs when a page is accessed and has dead tuples that are no longer visible to *any* active transaction.

Key Conditions for Pruning:

1. A prior `UPDATE` couldn’t find enough space on the page.
2. The data on the page exceeds the table’s `fillfactor` setting.



Page Pruning

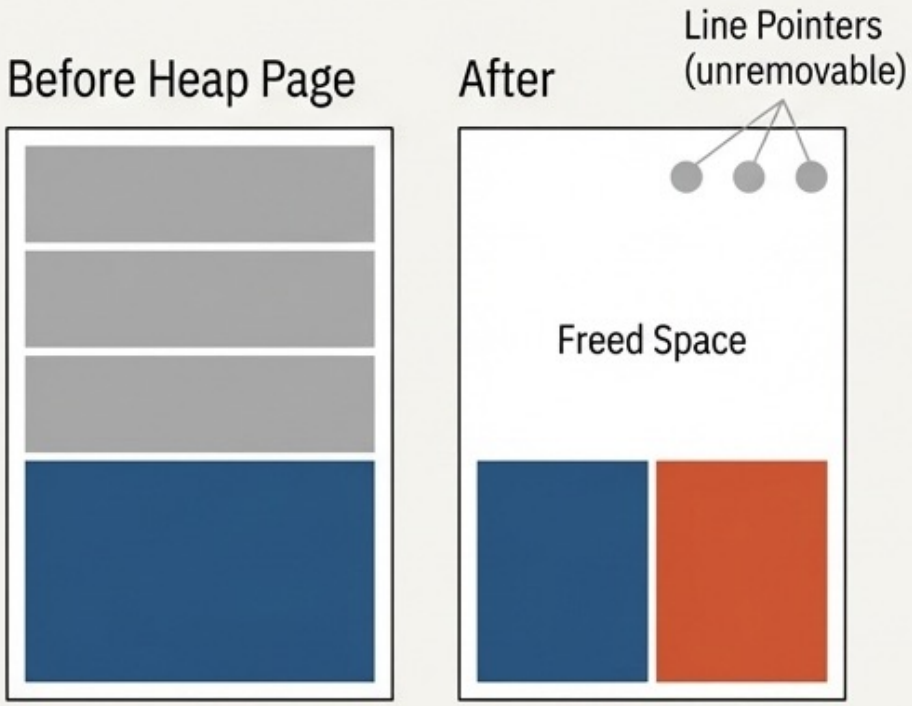


Full `VACUUM`

NotebookLM

Pruning Marks Dead Tuples, Freeing Space for New Updates

After another `UPDATE`, pruning is triggered. The space from the old tuples is reclaimed and used for the new version `(0,5)`. Notice the state of the old tuples changes from `normal` to `dead`. The line pointers `(0,1)`, `(0,2)`, `(0,3)` cannot be removed yet, as indexes may still point to them.




```
=> UPDATE hot SET s = 'E';
=> SELECT * FROM heap_page('hot',0);
 ctid  | state  |  xmin  | xmax
-------+--------+--------+------
 (0,1) | dead   |        |
 (0,2) | dead   |        |
 (0,3) | dead   |        |
 (0,4) | normal | 804 c  | 805
 (0,5) | normal | 804    | 0
 (0,5) | normal | 805    | 0 a
```

State changed from `normal` to `dead`, freeing up the physical space.

NotebookLM

But the Heap is Only Half the Story: Indexes are Still Bloated

Page Pruning does not touch the indexes. Every single version of the row, including those now marked `dead` in the heap, still has an entry in *every* index on the table. This is a major source of inefficiency.

`hot_s` index

```
=> SELECT * FROM index_page('hot_s',1);
 itemoffset | htid
------------+------
          1 | (0,1)
          2 | (0,2)
          3 | (0,3)
          4 | (0,4)
          5 | (0,5)
```

`hot_id` index

```
=> SELECT * FROM index_page('hot_id',1);
 itemoffset | htid
------------+------
          1 | (0,1)
          2 | (0,2)
          3 | (0,3)
          4 | (0,4)
          5 | (0,5)
```

Both indexes contain 5 entries, even though there is only one live row.

NotebookLM

Index Pointers are Marked `dead` Only When an Index Scan Encounters Them

When an index scan retrieves a pointer to a `dead` heap tuple, the server recognizes the tuple is gone. To prevent repeated lookups, it marks the *index item itself* as `dead`. This is an optimization, but the dead pointers still occupy space until a `VACUUM` can clean the index page.

```
=> EXPLAIN (analyze, costs off, timing off, summary off) SELECT * FROM hot WHERE id = 1;
                        QUERY PLAN
-----------------------------------------------------------
 Index Scan using hot_id on hot (actual rows=1 loops=1)
   Index Cond: (id = 1)
```

```
=> SELECT * FROM index_page('hot_id',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,1) | t
          2 | (0,2) | t
          3 | (0,3) | t
          4 | (0,4) | t
          5 | (0,5) | f
```

Flags flipped to `true` after the scan encountered the dead heap tuples.

Remains `false` because it points to the live tuple.

NotebookLM

This is still inefficient. What if the updated columns aren't even indexed?

The main performance problem remains:

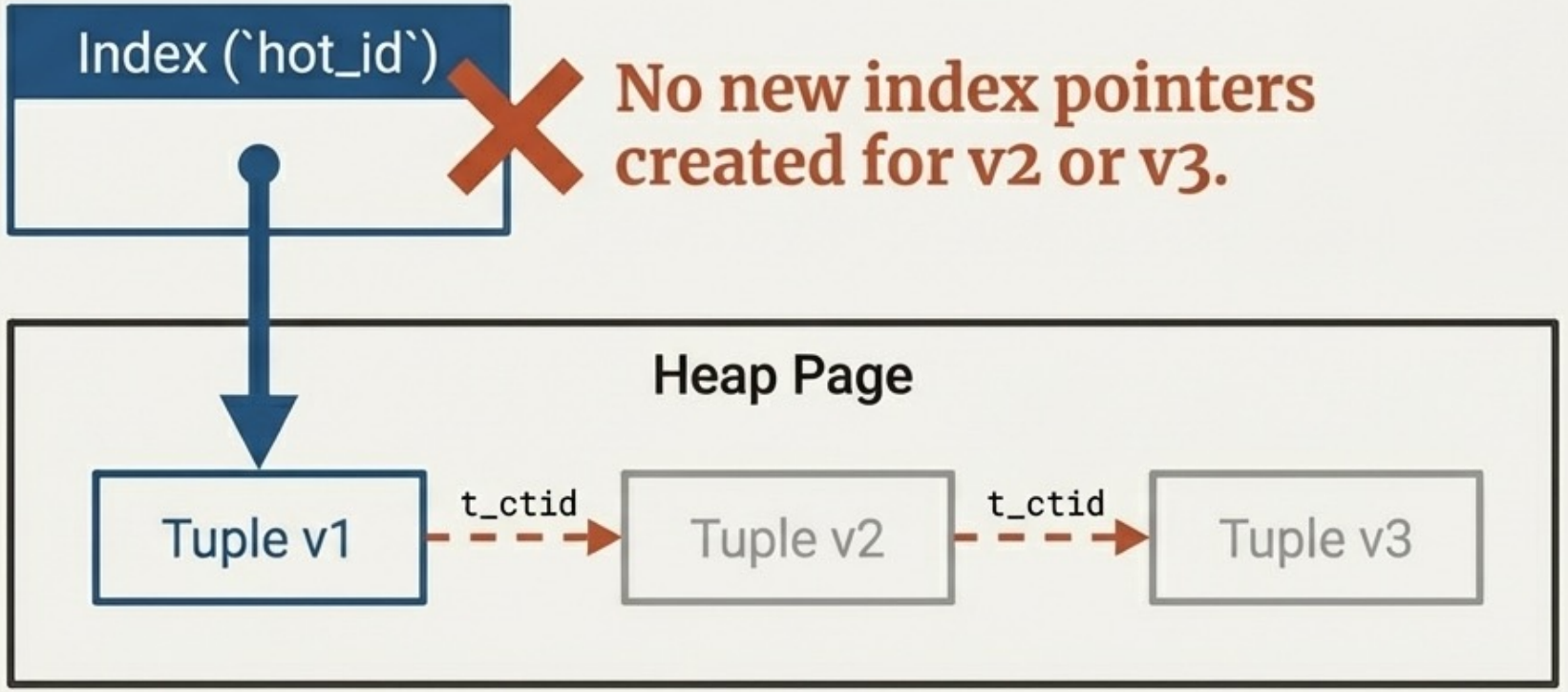
1. **Write Amplification**: Every `UPDATE` must modify *every index* on the table, even if the values in the indexed columns haven't changed.
2. **Index Bloat**: Indexes accumulate historical references to dead tuples, which require cleanup.

Can we avoid expensive index modifications for updates to non-indexed columns?

NotebookLM

The Solution: Heap-Only Tuple (HOT) Updates

When an `UPDATE` modifies only columns that are not part of any index, PostgreSQL can perform a highly **efficient HOT Update**. Instead of creating new index entries, the new tuple version is stored on the same heap page and chained directly to the previous version. The original index pointer remains unchanged, pointing to the start of this chain.



NotebookLM

The HOT Chain is Managed by Two Tuple Header Bits

Two flags in the tuple header (`t_infomask2`) orchestrate the HOT chain.



`HEAP_HOT_UPDATED` (hhu)

Set on an old tuple version. It tells an index scan, "This is not the latest version. Follow the `t_ctid` pointer to find the next tuple in the chain."

`HEAP_ONLY_TUPLE` (hot)

Set on a new tuple version. It signifies, "This tuple is part of a HOT chain and is not directly referenced by any index."

A HOT chain can only exist within a single data page.

NotebookLM

The HOT Chain in Action

After dropping the index on the updated column `s`, we repeat the updates. The output now clearly shows the chain forming.

Data as Evidence 1 (Heap)

```
=> SELECT * FROM heap_page('hot',0);
 ctid  | state  | … | hhu | hot | t_ctid
-------+--------+---+-----+-----+--------
(0,1)  | normal | … |  t  |     | (0,2)
(0,2)  | normal | … |  t  |  t  | (0,3)
(0,3)  | normal | … |  t  |  t  | (0,4)
(0,4)  | normal | … |     |  t  | (0,4)
```

Chain Head. `hhu` is set.
Chain Links. `hot` is set. Points to the next version.

Chain Tail. `hot` is set.

Data as Evidence 2 (Index)

```
=> SELECT * FROM index_page('hot_id',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,1) | f
```

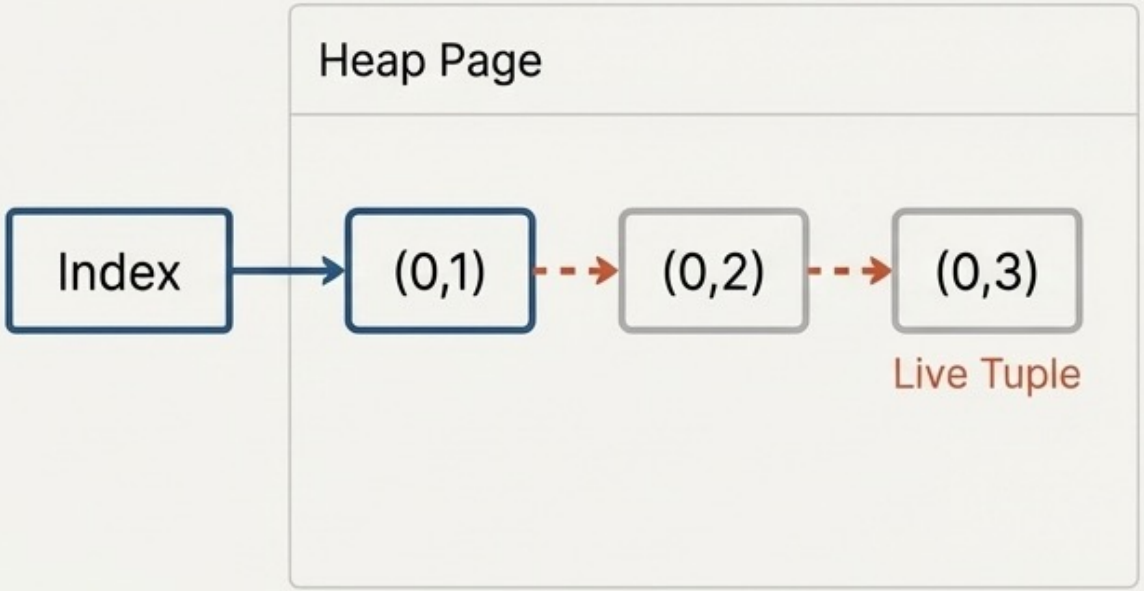
The `hot_id` index still has only ONE entry, pointing to the head of the chain.

NotebookLM

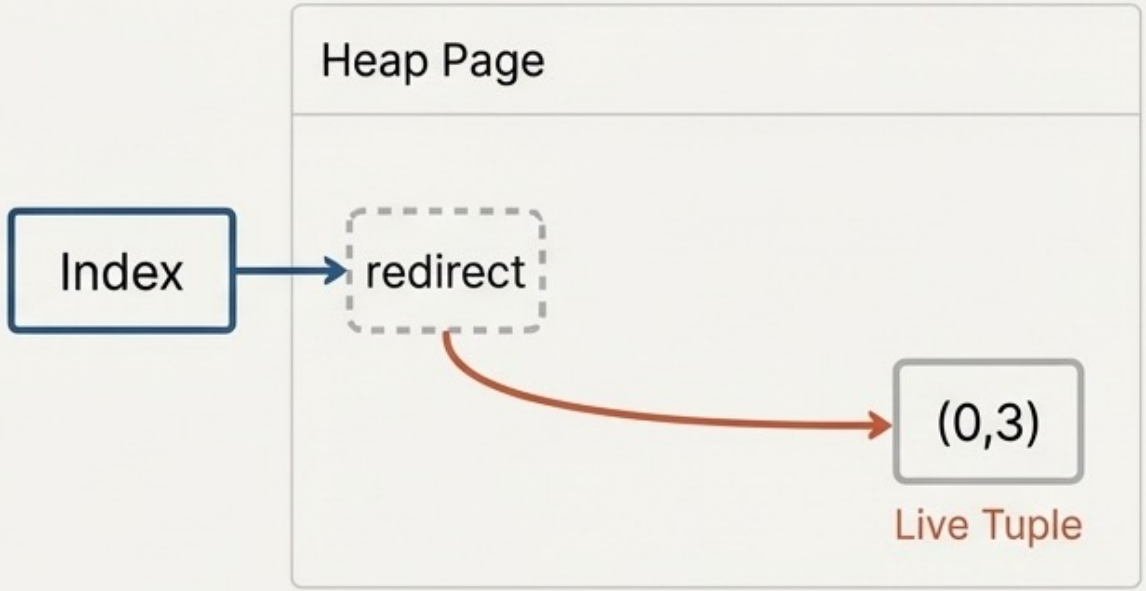
Pruning a HOT Chain is a Special Case

When a page with a HOT chain is pruned, intermediate links (marked `hot`) can be removed entirely because we know no indexes point to them. However, the chain head `(0,1)` *must* be preserved for the index. To do this without moving the tuple, its line pointer is converted to a `redirect` pointer, which points to the *new* head of the compacted chain.

Before



After Pruning



NotebookLM

Preserving the Index Entry Point via Redirection

An `UPDATE` triggers pruning. The old dead tuples `(0,2)` and `(0,3)` from the previous chain are now `unused`. The original head pointer `(0,1)` is now in a `redirect` state, pointing to the location of the current chain head.

```
=> UPDATE hot SET s = 'E';
=> SELECT * FROM heap_page('hot',0);
 ctid  |     state     | ... | hhu | hot | t_ctid
-------+---------------+-----+-----+-----+--------
 (0,1) | redirect to 4 |     |     |     |
 (0,2) | normal        | ... |     |  t  | (0,2)
 (0,3) | unused        |     |     |     |
 (0,4) | normal        | ... |  t  |  t  | (0,2)
```

Index scans hitting `(0,1)` are now redirected to the tuple at item offset 4.

The pointer for this old intermediate chain link is now free and marked `unused`.

NotebookLM

When The Page is Full: Breaking the HOT Chain

HOT chains can only exist within a single page. If an `UPDATE` occurs and there is no space on the current page for the new tuple version (and pruning cannot free enough), the chain must break. A new tuple is created on a different page, and a new entry must be added to the index to point to it, defeating the HOT optimization for that `UPDATE`.

Heap Page 0

```
-- heap_page('hot',0) output
(0,5) | normal | ... |      | t | (1,1)
```

`hhu` bit is not set. The chain breaks here. `t_ctid` now points to a new page (page 1).

Index

```
=> SELECT * FROM index_page('hot_id',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,1) | f
          2 | (1,1) | f
```

A second index entry is created, pointing to the head of the new chain on page 1.

NotebookLM

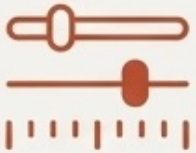
Practical Implications for Performance Tuning



Key Takeaway 1: Design for HOT Updates

`UPDATE`s are far cheaper when they are HOT.

Avoid including frequently-updated columns (e.g., `status`, `updated_at`, `counter`) in indexes if they are not critical for `WHERE` clauses.



Key Takeaway 2: Tune `fillfactor` to Manage Space

`fillfactor` reserves free space on each page for future `UPDATE`s.

For tables with frequent updates to non-indexed columns, **lowering** `fillfactor` (e.g., to 80-90%) gives HOT chains more room to grow within a page.

This is a trade-off: It improves `UPDATE` performance and reduces index bloat at the cost of increased table size.

Understanding these mechanics allows you to design tables and indexes that work **with** PostgreSQL's internal logic, not against it.

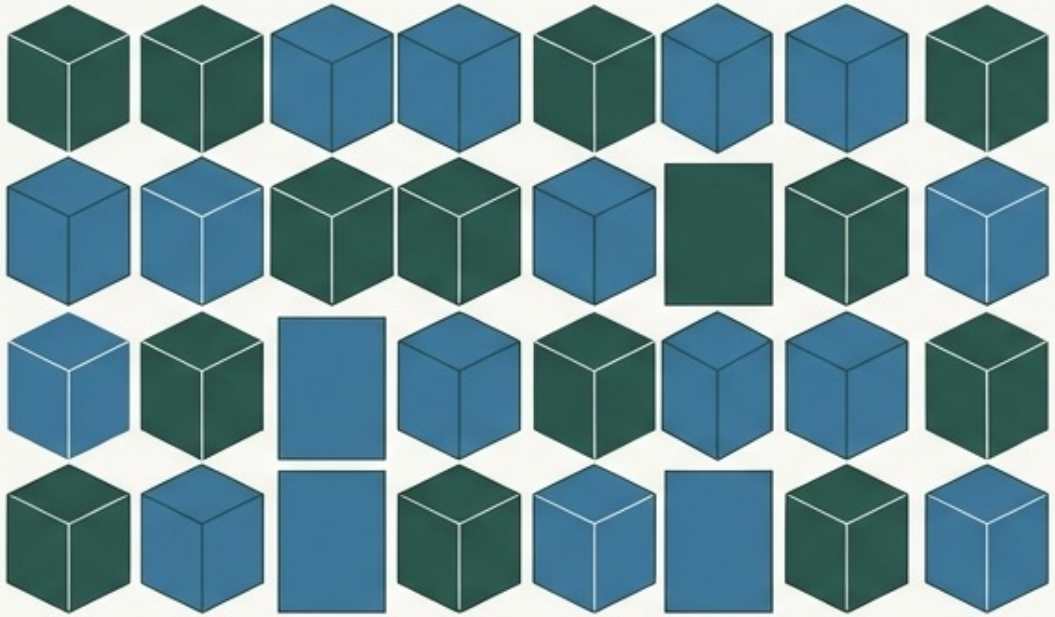
NotebookLM

6. Vacuum and Autovacuum

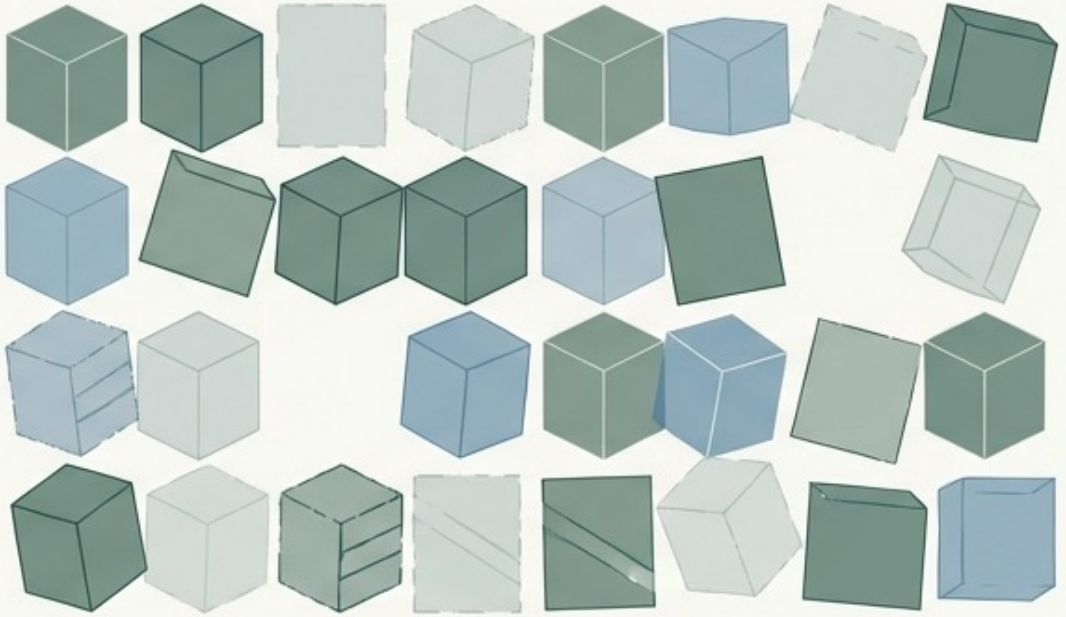
The Art of Database Hygiene: Mastering VACUUM in PostgreSQL

How PostgreSQL's essential maintenance process prevents bloat and ensures peak performance.

Healthy Database



Bloated Database



PostgreSQL's MVCC architecture is powerful, but it creates 'digital clutter'—obsolete row versions called dead tuples. Left unchecked, this clutter leads to performance degradation and wasted storage. `VACUUM` is the tool that cleans it up.

NotebookLM

How ‘Dead Tuples’ Are Born

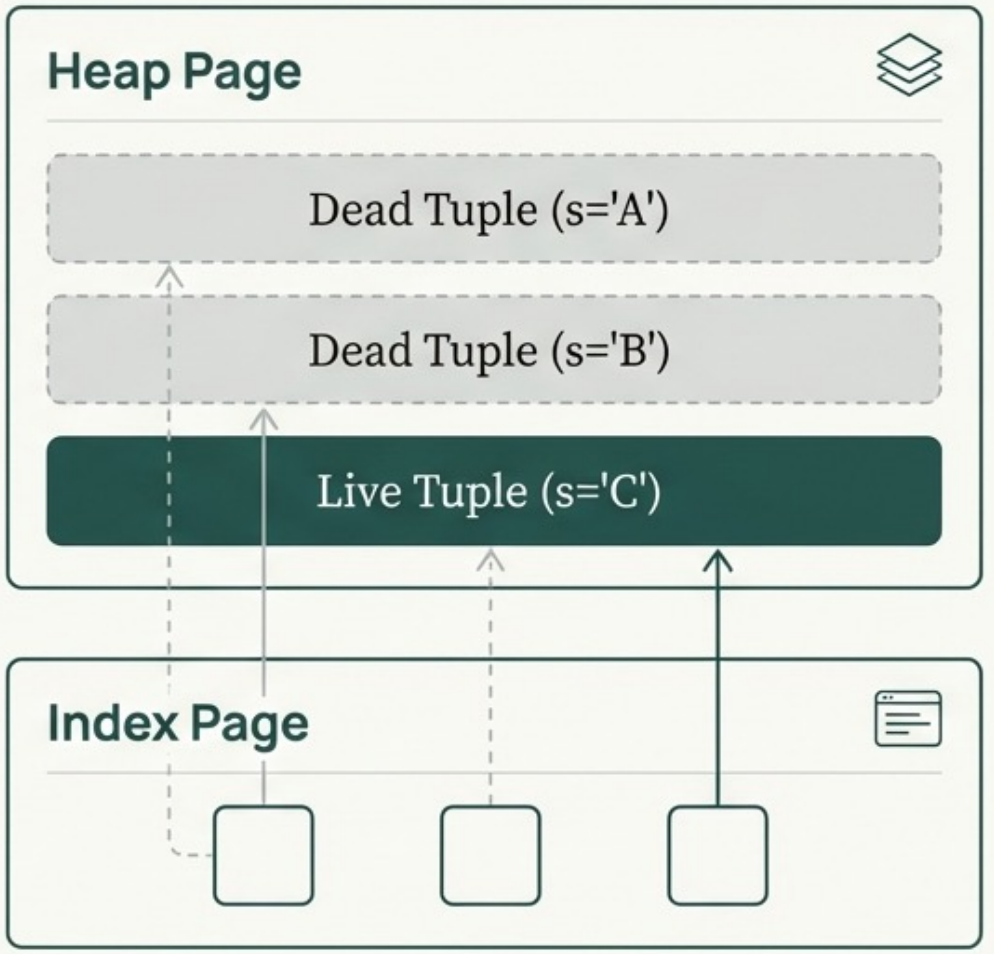
Every UPDATE or DELETE in PostgreSQL doesn’t overwrite data. Instead, it marks the old row version as “dead” and creates a new one. Let’s watch it happen.

```
-- 1. We create a table and insert one row.
CREATE TABLE vac(id integer, s char(100))
   WITH (autovacuum_enabled = off);
CREATE INDEX vac_s ON vac(s);

INSERT INTO vac(id,s) VALUES (1,'A');

-- 2. Two updates create two dead tuples.
UPDATE vac SET s = 'B';
UPDATE vac SET s = 'C';
```

The Before State



NotebookLM

The Cleanup Crew: How VACUUM Reclaims Space

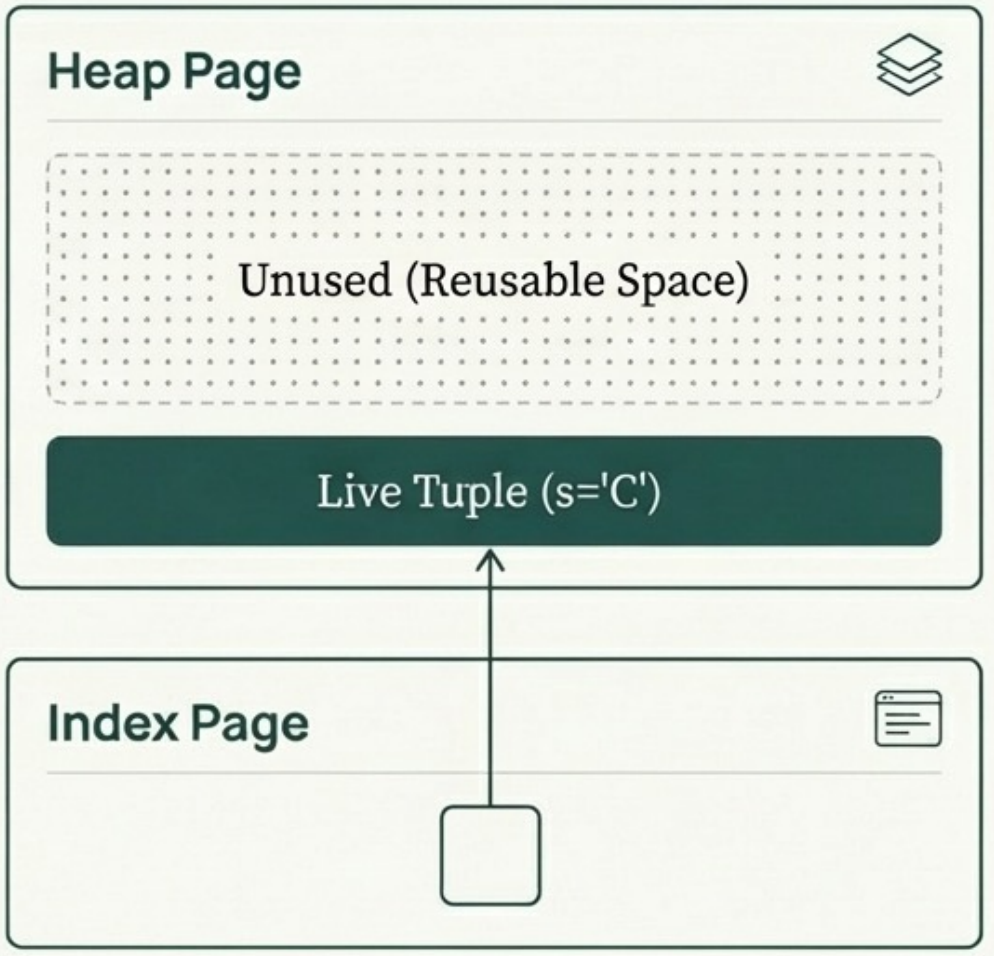
`VACUUM` scans the table and its indexes to remove dead tuples, making the space available for new rows.

The Command

```
VACUUM vac;
```

The dead tuples and their corresponding index entries are removed. The space they occupied is now free to be reused by future `INSERT` or `UPDATE` operations.

The After State



NotebookLM

The Complication: Why VACUUM Can't Always Clean Up

`VACUUM` must honor transaction visibility. It cannot remove a dead tuple if an older, concurrent transaction might still need to "see" it. This point-in-time is determined by the `oldest xmin`.




```
INFO: table "vac": found 0
removable, 2 nonremovable row
versions in 1 out of 1 pages
DETAIL: 1 dead row versions
cannot be removed yet, oldest
xmin: 834
```

One is the current live version, the other is the dead version protected by the open transaction.

The transaction ID of the oldest active transaction. `VACUUM` cannot remove anything created after this point.

Long-running transactions are a primary cause of database bloat because they prevent `VACUUM` from doing its job effectively.

NotebookLM

The Anatomy of a VACUUM Operation



Stage 1: Heap Scan

Scans table pages (using the Visibility Map to skip all-visible pages). Collects IDs (TIDs) of dead tuples into a memory buffer (`maintenance_work_mem`).



Stage 2: Index Vacuum

Scans all of the table's indexes. Removes index entries that point to the dead TIDs collected in Stage 1.



Stage 3: Table Vacuum

Re-scans the heap pages. Marks the space occupied by the dead tuples (whose index entries are now gone) as reusable. Updates the Free Space Map.



Stage 4: Table Truncation

(Optional) If there are empty pages at the file's end, `VACUUM` may lock the table briefly to truncate the file and return disk space to the OS.

NotebookLM

The Automation Challenge: How Often is “Often Enough”?



Too Infrequently

- Table and index bloat.
- Poor query performance.
- Huge cleanup jobs when `VACUUM` finally runs, potentially requiring multiple passes over indexes.

Too Frequently

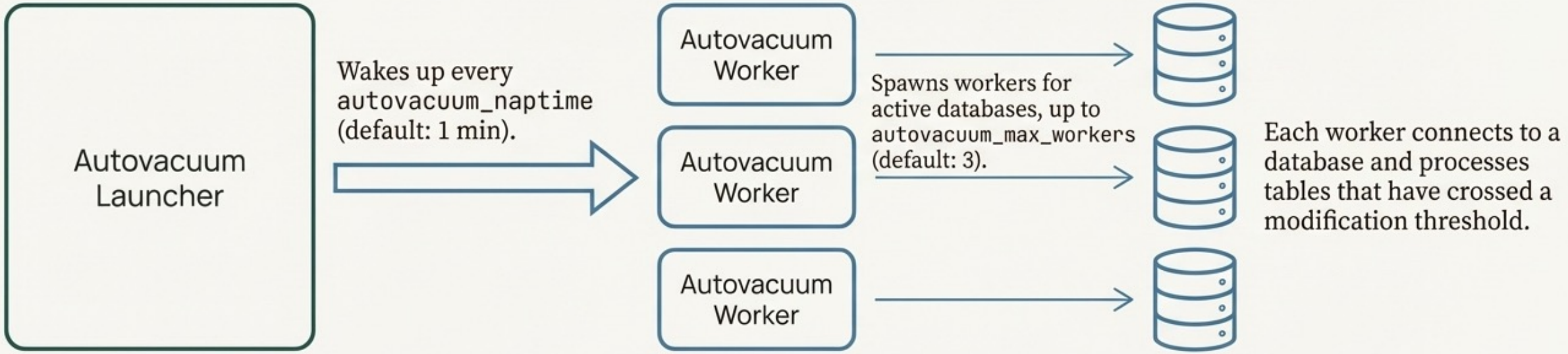
- Wasted CPU and I/O cycles on maintenance instead of useful work.

A fixed schedule is inefficient because workloads change. How can we make vacuuming responsive to the actual rate of change in the database?

NotebookLM

The Solution: The Autovacuum Daemon

PostgreSQL's built-in `autovacuum` daemon automates `VACUUM` and `ANALYZE` operations based on table activity, not a fixed schedule.



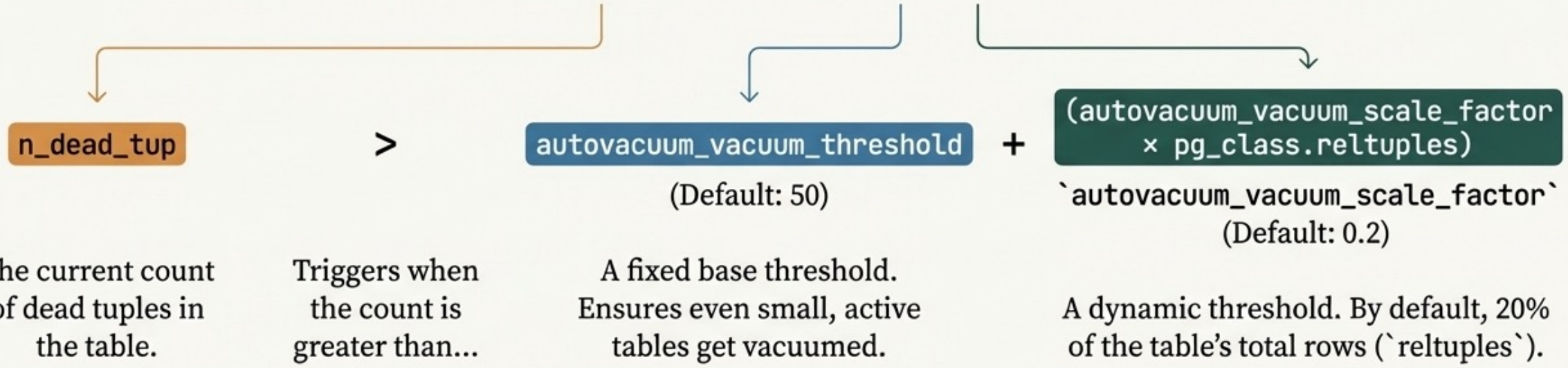
Key Features

- Activity-Based: Triggers based on the number of dead or modified rows.
- Throttled: Designed to run in the background with minimal performance impact.
- Configurable: Highly tunable at both the global and table level.

NotebookLM

The Rules of Engagement: When Autovacuum Triggers

```
pg_stat_all_tables.n_dead_tup > autovacuum_vacuum_threshold +
    (autovacuum_vacuum_scale_factor × pg_class.reltuples)
```



The default 20% scale factor means a table with 1 million rows needs 200,050 dead tuples to trigger a vacuum. For very large tables, this can be too high, leading to bloat. It's one of the most important parameters to tune.

NotebookLM

Beyond Deletes: Vacuuming for Inserts and Freezing

Tables with only `INSERT`s have no dead tuples. So why do they still need vacuuming?

The Answer

1. **Transaction ID Freezing:** To prevent transaction ID wraparound failure (a catastrophic event).
2. **Updating the Visibility Map:** Marking pages as 'all-visible' is crucial. This allows PostgreSQL to perform fast **Index-Only Scans**, avoiding heap fetches and dramatically improving query performance.

The Trigger Formula for Inserts

```
pg_stat_all_tables.n_ins_since_vacuum > autovacuum_vacuum_insert_threshold + (autovacuum_vacuum_insert_scale_factor × pg_class.reltuples)
```

Default values:

- `autovacuum_vacuum_insert_threshold`: `1000`
- `autovacuum_vacuum_insert_scale_factor`: `0.2`

Even if you only insert data, **autovacuum** is essential for long-term stability and query performance.

NotebookLM

Keeping Statistics Fresh with ANALYZE



VACUUM cleans the house



ANALYZE takes inventory

It collects statistics about data distribution that the query planner uses to create efficient execution plans.

How Autovacuum Handles It

Autovacuum also triggers ANALYZE automatically based on the number of changed (inserted, updated, or deleted) rows.

The ANALYZE Trigger Formula

`pg_stat_all_tables.n_mod_since_analyze > autovacuum_analyze_threshold + (autovacuum_analyze_scale_factor × pg_class.reltuples)`

Defaults

- `autovacuum_analyze_threshold`: 50
- `autovacuum_analyze_scale_factor`: 0.1 (10%)

Key Takeaway: Out-of-date statistics are a leading cause of suddenly slow queries. Autovacuum's auto-analyze feature is your first line of defense.

NotebookLM

Fine-Tuning the Machine: Per-Table Configuration

The Principle: Global settings are a starting point, but optimal performance requires tuning autovacuum for your most important or problematic tables.

The Method: Use `ALTER TABLE ... SET` to override global defaults with storage parameters.

For a very large, high-traffic OLTP table	For a TOAST table of a large text column
`ALTER TABLE public.orders SET (autovacuum_vacuum_scale_factor =0.02, -- Vacuum at 2% bloat, not 20% autovacuum_analyze_scale_factor = 0.01); -- Analyze at 1% change`	`ALTER TABLE public.articles SET (toast.autovacuum_vacuum_cost_delay = 0); -- Make TOAST vacuum more aggressive`

Key Parameters for Table-Level Tuning

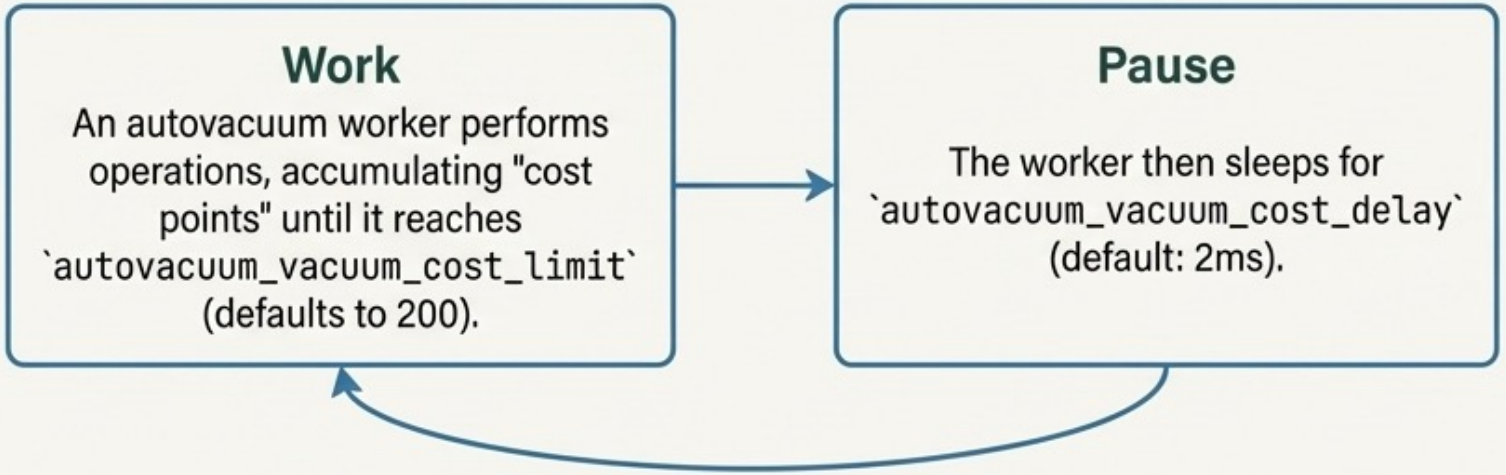
- `autovacuum_vacuum_threshold` / `scale_factor`
- `autovacuum_vacuum_insert_threshold` / `scale_factor`
- `autovacuum_analyze_threshold` / `scale_factor`
- `autovacuum_vacuum_cost_delay` / `cost_limit`
- (And their toast. equivalents for TOAST tables)

NotebookLM

Managing the Impact: Cost-Based Throttling

Core Concept: To avoid disrupting your primary workload, autovacuum uses a cost-based throttling system. It does a small amount of work, then pauses.

The Mechanism



How Costs are Calculated

- Reading a page from cache (`vacuum_cost_page_hit`): **1 point**
- Reading a page from disk (`vacuum_cost_page_miss`): **2 points**
- Marking a clean page as dirty (`vacuum_cost_page_dirty`): **20 points**

Important Note

The `cost_limit` is shared among all active workers. To make autovacuum more aggressive overall, you must increase both `autovacuum_max_workers` and `autovacuum_vacuum_cost_limit`.

Historical Context

Before PostgreSQL 12, the default delay was 20ms, which was often too slow for modern hardware. The current 2ms default is much more effective.

NotebookLM

Diagnosis: When `maintenance_work_mem` Isn't Enough

The Scenario

What happens when `VACUUM` finds more dead tuples than can fit in its memory buffer (`maintenance_work_mem`)?

The Consequence

`VACUUM` is forced to perform multiple passes. It scans the heap, fills its buffer, vacuums the indexes for those tuples, then repeats the entire process. This can dramatically increase runtime and I/O.

How to Fix It

1. **Increase `maintenance_work_mem`:** Give `VACUUM` more memory to work with.
2. **Vacuum More Frequently:** Tune autovacuum to run more often, so each run has fewer dead tuples to process.

The Evidence

```
INFO:  vacuuming "public.vac"
INFO:  scanned index "vac_s" to remove 174522 row versions
DETAIL:  CPU: user: 0.02 s, system: 0.00 s, elapsed: 0.05 s
INFO:  table "vac": removed 174522 dead item identifiers in
3009 pages
DETAIL:  CPU: user: 0.00 s, system: 0.01 s, elapsed: 0.07 s
INFO:  scanned index "vac_s" to remove 174522 row versions
DETAIL:  CPU: user: 0.02 s, system: 0.00 s, elapsed: 0.05 s
INFO:  table "vac": removed 174522 dead item identifiers in
3009 pages
DETAIL:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.01 s
INFO:  scanned index "vac_s" to remove 150956 row versions
DETAIL:  CPU: user: 0.02 s, system: 0.00 s, elapsed: 0.04 s
INFO:  table "vac": removed 150956 dead item identifiers in
2603 pages
DETAIL:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO: index "vac_s" now contains 500000 row versions in
932 pages...
```

Pass 1

Pass 2

Pass 3

Confirmation from Logs:

```
LOG:  ... index scans: 3
```

NotebookLM

Your Toolkit for Monitoring Autovacuum

Tool 1: The Server Logs (Your Best Friend)

How to Enable

```
ALTER SYSTEM SET log_autovacuum_min_duration = 0;
```

Logs all autovacuum actions. Set to a higher value (e.g., '10s') to only log long-running ones.

What to Look For

should be 1!

```
LOG:  automatic vacuum of table "internals.public.vac": index scans: 3
    pages: 0 removed, 17242 remain...
    tuples: 500000 removed, 500000 remain...
    buffer usage: 46038 hits, 5670 misses, 6770 dirtied
    system usage: CPU: user: 0.31 s, system: 0.33 s, elapsed: 3.57 s
```

Tool 2: Live Progress Views

What it is

The `pg_stat_progress_vacuum` view lets you see what `VACUUM` (manual or auto) is doing *right now*.

Key Columns to Watch

`phase`	Shows the current stage (e.g., `` `scanning heap` ``, `` `vacuuming indexes` ``).
`heap_blks_scanned` vs. `heap_blks_total`	Tracks progress through the table.
`index_vacuum_count`	The number of index scan passes.

NotebookLM

The Health Check: Is Autovacuum Keeping Up?

The Core Question

How can you tell if your autovacuum tuning is aggressive enough for your workload?

Regularly check which tables are currently considered "due" for vacuuming or analyzing. If this list is consistently long or growing, your settings are too conservative.

Actionable Advice

If tables are consistently falling behind (`n_dead_tup` is far greater than `max_dead_tup`), it's time to act. Your primary levers are:

1. **Decrease** `autovacuum_vacuum_cost_delay`.
2. **Increase** `autovacuum_vacuum_cost_limit`.
3. **Increase** `autovacuum_max_workers`.

The Query

```
-- A view to see which tables need vacuuming and why
CREATE VIEW need_vacuum AS
  WITH c AS (
    ...
    ) -- Simplified for slide
  )
  SELECT st.schemaname || '.' || st.relname AS tablename,
         st.n_dead_tup AS dead_tup,
         (c.threshold + c.scale_factor * c.reltuples) AS
         max_dead_tup,
         st.last_autovacuum
  FROM  pg_stat_all_tables st JOIN c ON c.oid = st.relid
  WHERE st.n_dead_tup > (c.threshold + c.scale_factor *
         c.reltuples);
```

NotebookLM

7. Freezing

On a busy system generating 1,000 transactions per second, your PostgreSQL database will run out of Transaction IDs in about 6 weeks.



This introduces the core problem: **TXID Wraparound**.

PostgreSQL uses a 32-bit integer for Transaction IDs. This provides a finite space of about 4 billion unique IDs.

Once this number is exhausted, the counter must reset or "wrap around."

However, simple wraparound is catastrophic. A new transaction with a low ID (e.g., TXID 100) would suddenly appear "older" than a recent transaction with a high ID (e.g., TXID 4 billion), breaking the fundamental rules of transaction visibility and leading to data corruption.

This presentation explains the elegant, multi-layered system PostgreSQL uses to prevent this disaster: **Freezing**.

NotebookLM

The 8-Byte Question: Why Not 64-bit TXIDs?

The choice of a 32-bit TXID is a critical design decision balancing longevity against performance.

24+ Bytes



Optimized for Performance

32+ Bytes



System-Wide Bloat

The Cost of Storage:

Every single row version (tuple) in PostgreSQL stores two TXIDs in its header: `xmin` (the transaction that inserted it) and `xmax` (the transaction that deleted it).

Using 64-bit TXIDs would add 8 extra bytes of overhead *per tuple version*.

The System-Wide Impact:

This seemingly small increase would **bloat every table**, leading to:

- Increased memory usage for data in the buffer cache.
- Increased I/O for reading/writing data from disk.

The result is a tangible decrease in overall system efficiency.

The Compromise:

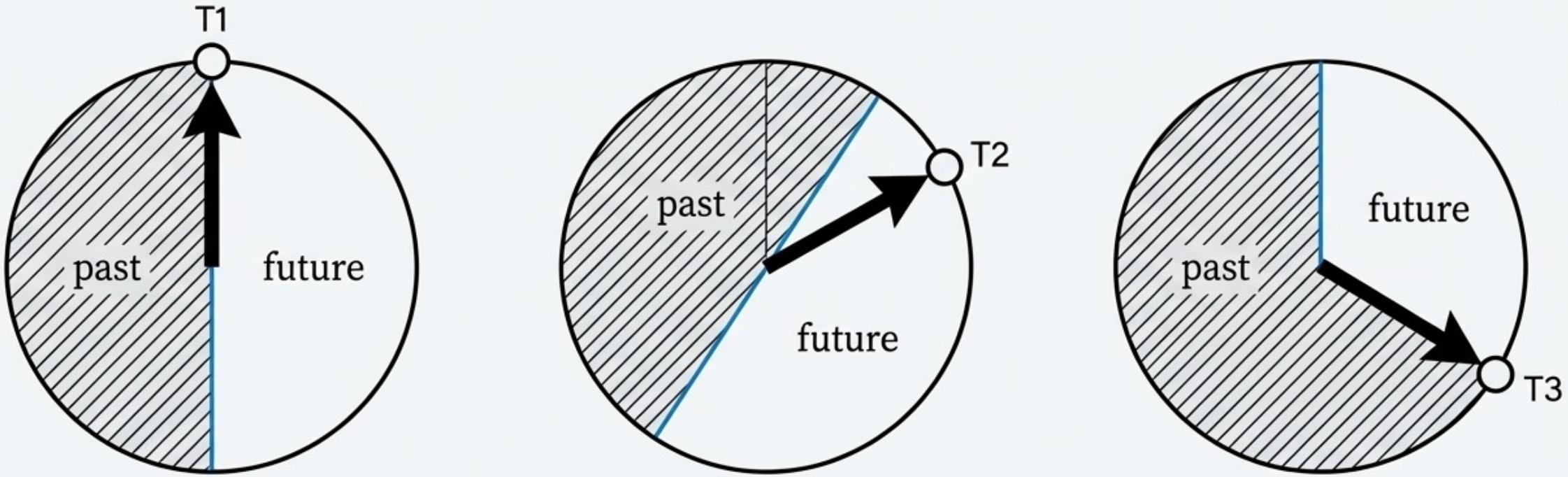
PostgreSQL does use a 64-bit `FullTransactionId` internally (combining the 32-bit ID with a 32-bit 'epoch'), but this is never stored on data pages to maintain efficiency. The on-disk format is optimized for performance.

NotebookLM

Thinking in Circles: The Clock Analogy

To solve the wraparound problem, PostgreSQL compares transaction **age**, not the absolute value of their IDs. Instead of “larger” or “smaller,” it thinks in terms of “younger” or “older.”

This comparison is performed efficiently using 32-bit modular arithmetic.



For any given transaction (the “hand” of the clock), the half of the circle in front of it is the “future” (younger transactions), and the half behind it is the “past” (older transactions).

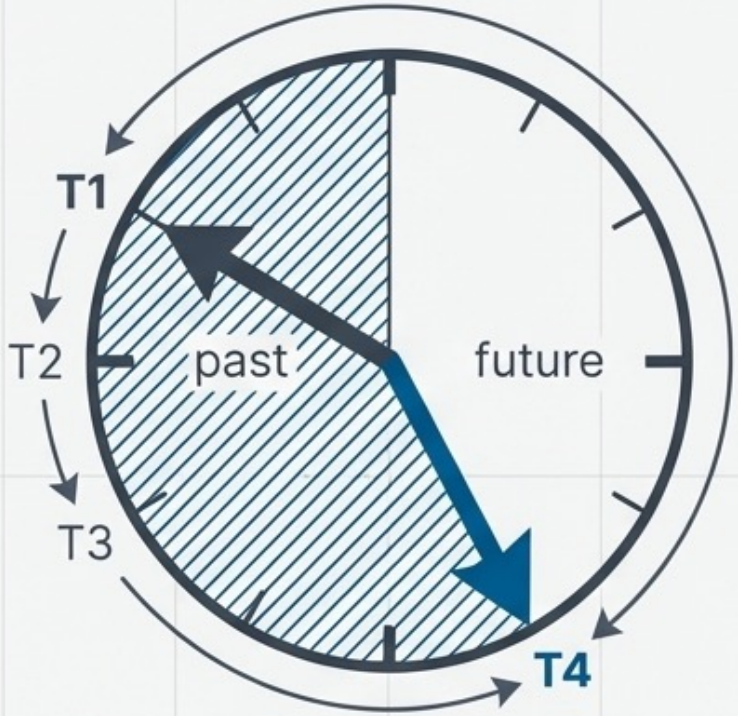
NotebookLM

The Clock Breaks: When the Past Becomes the Future

The clock analogy is powerful, but it has a critical weakness. A transaction can become *too old*.

The Wraparound Danger

- Consider a very old, committed transaction, T1.
- If more than 2 billion new transactions occur (half the clock face), the current transaction's "present" will move past T1.
- From the perspective of these new transactions, T1 is now in their "future."



The Consequence: Data Invisibility

A transaction that sees T1 in its "future" will treat it as if it hasn't happened yet. The data created by T1 becomes invisible, effectively disappearing from the database for all new transactions. This is a catastrophic data integrity failure.

The database must prevent any transaction from becoming this "ancient."

NotebookLM

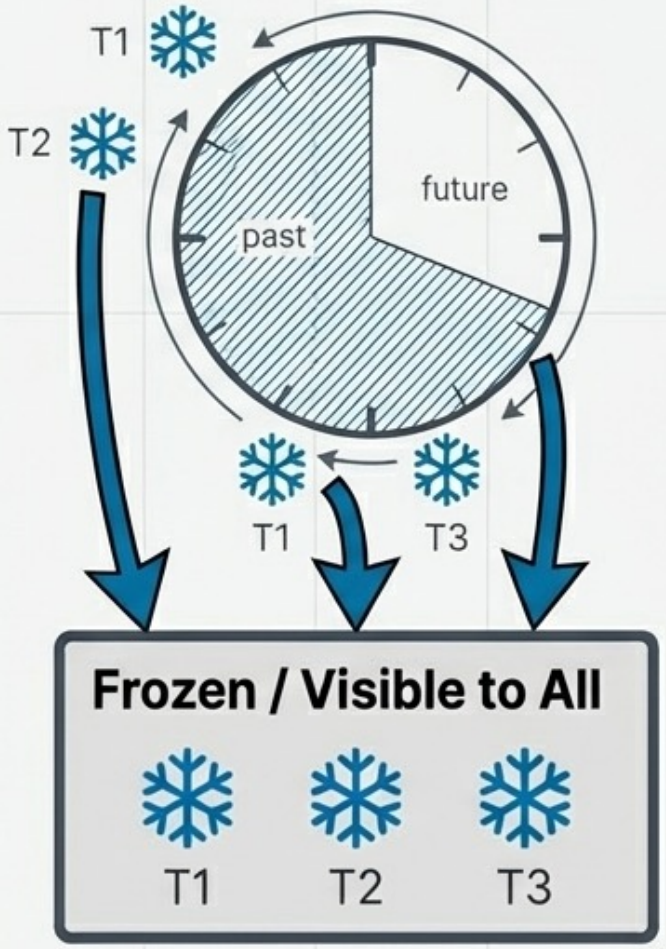
The Solution: Freezing Tuples in Ancient History

To prevent the time travel problem, PostgreSQL VACUUM performs an essential task called **Freezing**.



What is a Frozen Tuple?

- A frozen tuple is marked in a special way to signify it was created by a transaction so far in the past that it is considered "ancient history."
- Conceptually, its `xmin` value is treated as "negative infinity."



The Rule Change

- For frozen tuples, the normal visibility rules are suspended. They are considered visible to **all** current and future transactions, regardless of their snapshot.
- This permanently locks their state as "visible," resolving the wraparound ambiguity. The clock comparison for their `xmin` is no longer necessary.

The Benefit

- Once all tuples created by a given TXID are frozen, that TXID can be safely reused after the counter wraps around.

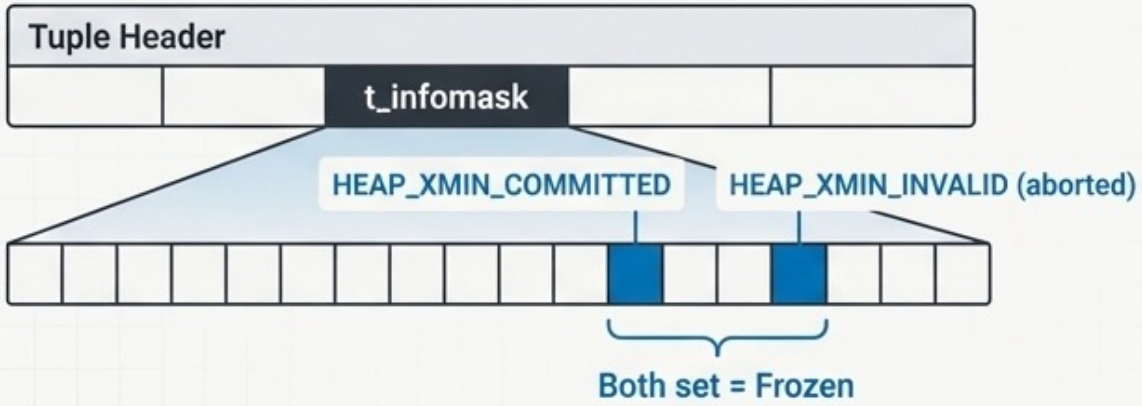
NotebookLM

Under the Hood: How Freezing Actually Works

Freezing isn't done by changing the xmin value itself anymore. The original TXID is preserved for debugging.

The Modern Method (PostgreSQL 9.4+):

- The frozen state is stored using a combination of two hint bits in the tuple's header (t_infomask).
- When both the 'committed' and 'aborted' hint bits are set for xmin, the tuple is considered frozen.



The Legacy Method:

- Older versions of PostgreSQL would physically replace xmin with the special value 2 (FrozenTransactionId). You may still see this value in documentation or on upgraded systems.

Code Example: Inspecting Tuples:

The pageinspect extension allows you to see these properties directly. The heap_page function below decodes the hint bits to show the tuple state ('c' for committed, 'f' for frozen).

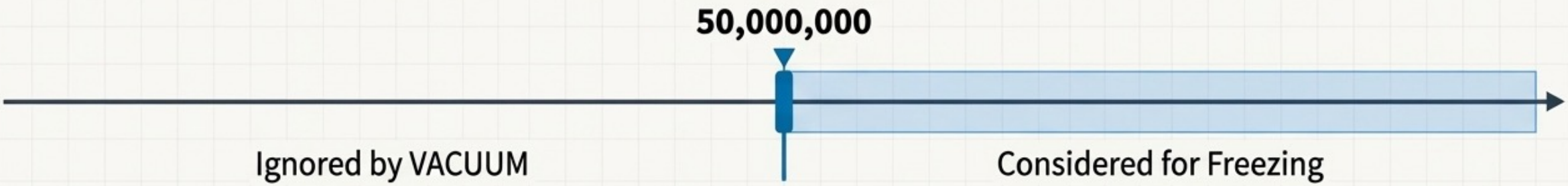
```
-- Simplified excerpt from source function
SELECT
  t_xmin || CASE
    WHEN (t_infomask & 256+512) = 256+512 THEN ' f' -- Both bits
    WHEN (t_infomask & 256) > 0 THEN ' c'           -- Committed
                                                       bit set

    -- ...
  END AS xmin_state,
  age(t_xmin) AS xmin_age
FROM heap_page_items(...);
```

NotebookLM

Layer 1: The First Threshold (`vacuum_freeze_min_age`)

Freezing is controlled by a series of cascading age-based parameters. The first is `vacuum_freeze_min_age`.



Purpose: This parameter tells VACUUM when to start considering tuples for freezing.

Default Value: 50,000,000 transactions.

How it Works:

- During a normal VACUUM, if a tuple's `xmin` is older than `vacuum_freeze_min_age`, VACUUM will freeze it.
- Tuples younger than this age are ignored to avoid unnecessary work.

Tuning Considerations:

- **Too Low**: Wastes resources by repeatedly freezing new versions of frequently updated ('hot') rows. This causes unnecessary write amplification.
- **Too High**: Delays freezing work, potentially allowing a large backlog of unfrozen tuples to accumulate.

This is the system's routine, day-to-day freezing mechanism. However, it has a significant blind spot.

NotebookLM

The Efficiency Trap: The Visibility Map Blind Spot

A standard VACUUM relies on an optimization called the **Visibility Map** (VM) to avoid scanning entire tables.

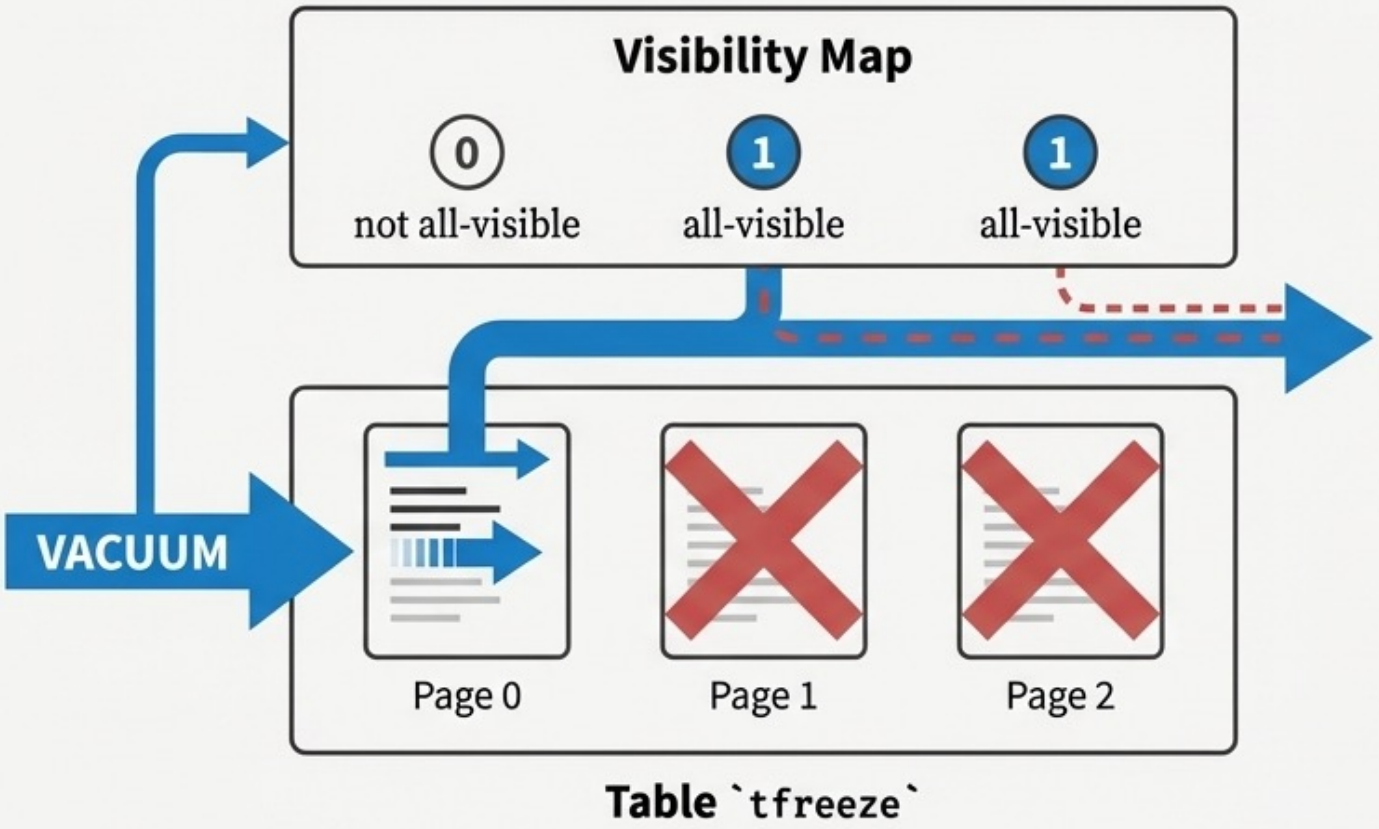


Table `tfreeze`

How the VM Works

- The VM tracks which data pages contain only tuples that are visible to all current and future transactions. These pages are marked as 'all-visible.'
- VACUUM can safely **skip** scanning these pages, dramatically reducing I/O and speeding up the process.

The Problem

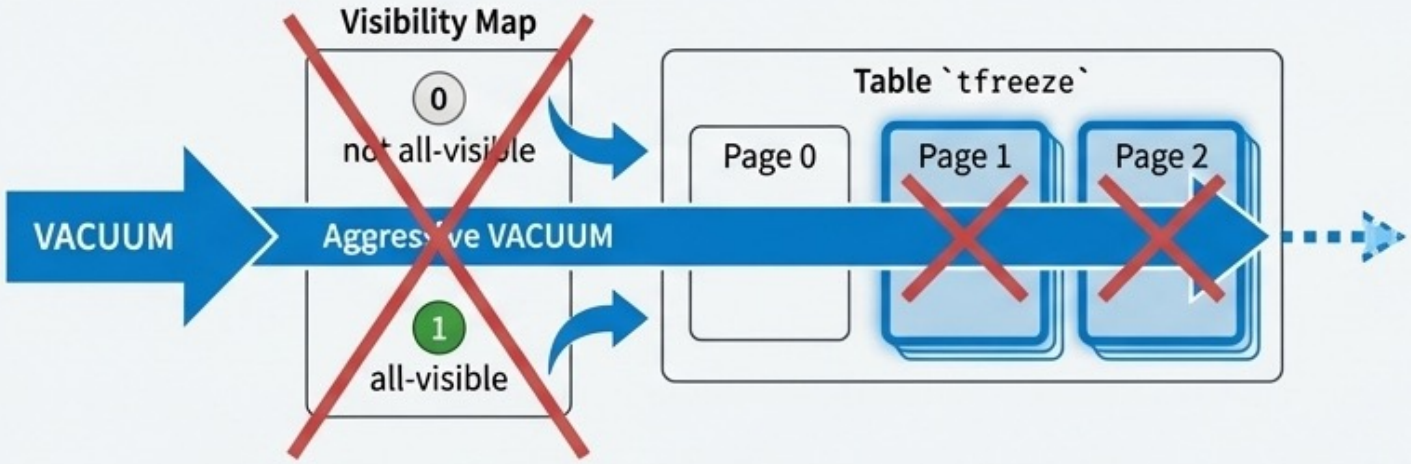
- If a page is marked 'all-visible,' VACUUM will not visit it.
- This means that even if the page contains tuples older than `vacuum_freeze_min_age`, they will **not be frozen**.
- An UPDATE or DELETE on a page will clear its 'all-visible' bit, allowing VACUUM to process it again. But for read-only or rarely updated tables, old tuples can remain unfrozen indefinitely.

Example: The source shows that after an UPDATE on page 0, its `all_visible` bit is cleared. A subsequent VACUUM freezes a tuple on page 0 but completely skips page 1, which remained `all_visible`.

NotebookLM

Layer 2: Aggressive Freezing (vacuum_freeze_table_age)

To overcome the Visibility Map's blind spot, PostgreSQL employs a more forceful strategy called aggressive vacuuming.



The Trigger

This is controlled by `vacuum_freeze_table_age`.

- Default Value: 150,000,000 transactions.

The Metric: `relfrozenxid`

The `pg_class` system catalog stores a `relfrozenxid` for each table. This is a watermark: it guarantees all transactions older than this ID have been frozen within that table.

How it Works

- When the age of a table's `relfrozenxid` (`age(relfrozenxid)`) exceeds `vacuum_freeze_table_age`, VACUUM initiates an aggressive scan.
- During an aggressive scan, VACUUM ignores the Visibility Map and scans every single page of the table, ensuring all old tuples are found and frozen.
- The Freeze Map is used during this process to avoid re-scanning pages that are already known to be all-frozen, making the process more efficient and resumable if interrupted.

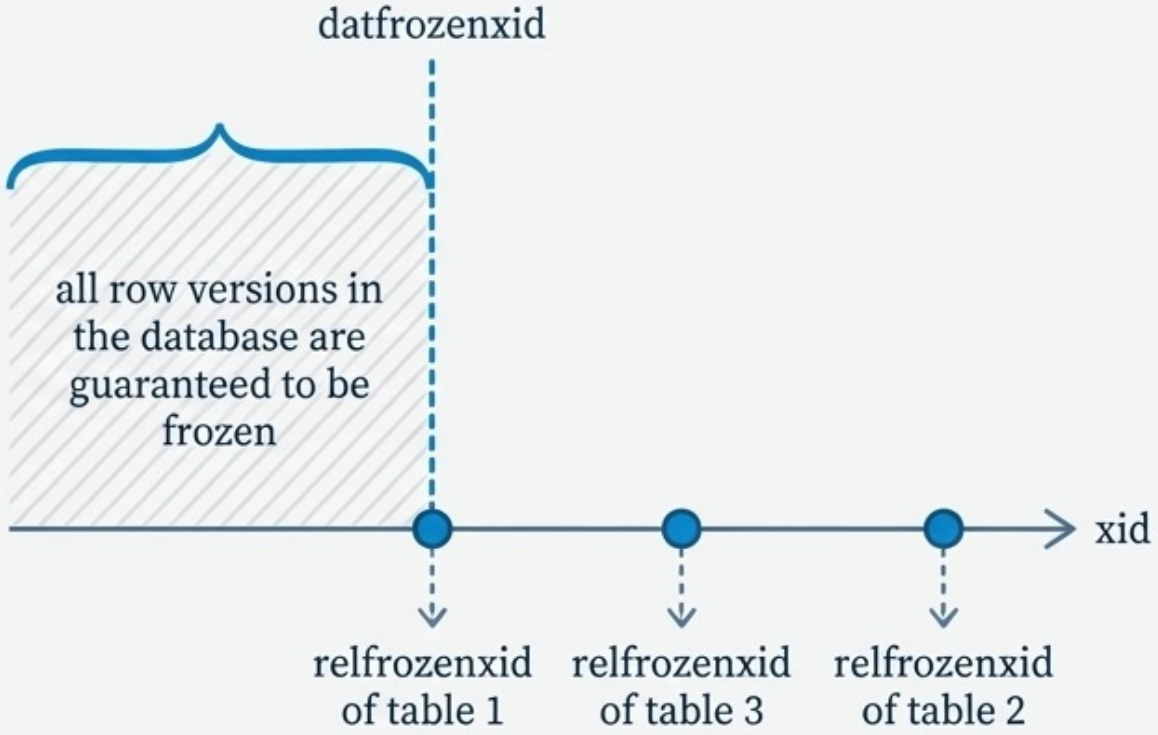
NotebookLM

The Big Picture: `relfrozenxid` and `datfrozenxid`

PostgreSQL tracks freezing progress at both the table and database level.

Definitions:

- **`relfrozenxid` (per Table):** The oldest transaction ID that might still have unfrozen tuples *in that specific table*.
- **`datfrozenxid` (per Database):** The oldest transaction ID that might still have unfrozen tuples *anywhere in the entire database*. It is effectively the **`MIN(relfrozenxid)`** across all tables in that database.

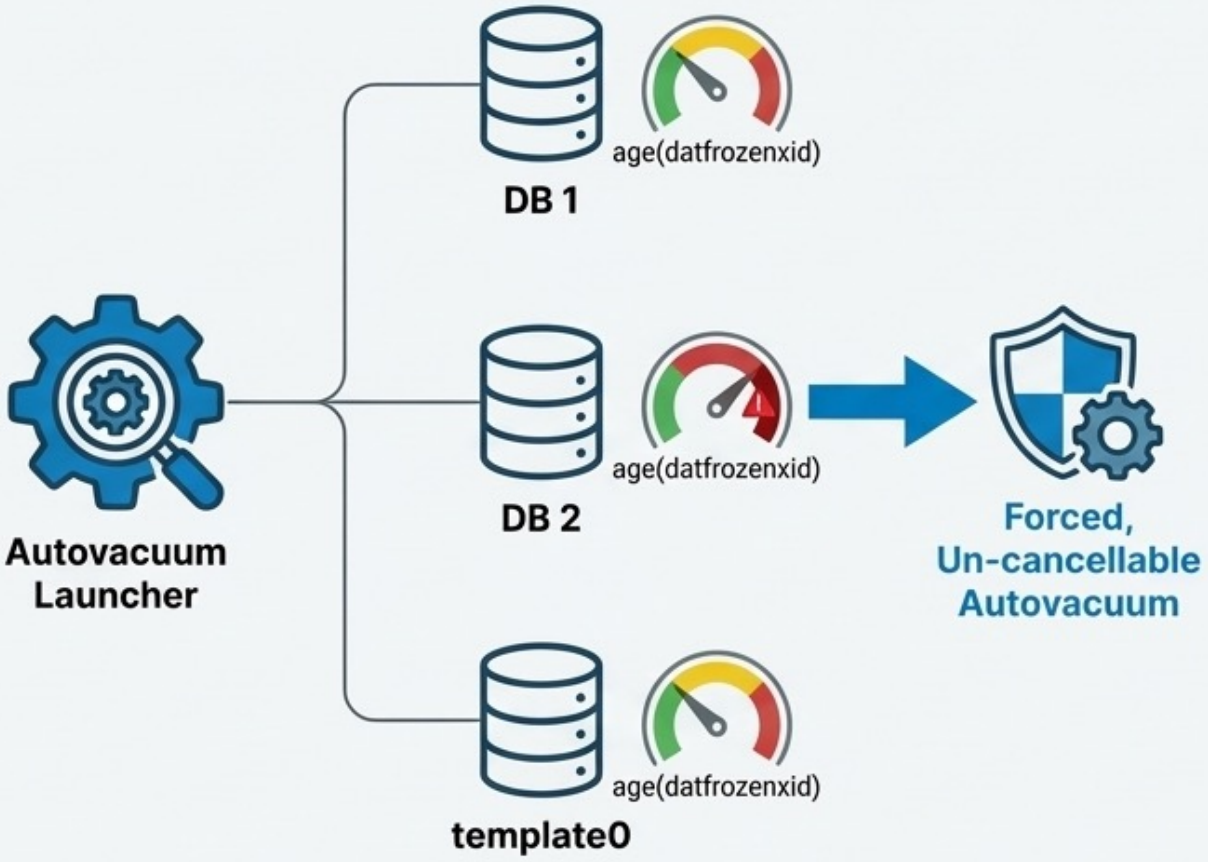


This **`datfrozenxid`** serves as the ultimate safety metric for the entire database.

NotebookLM

Layer 3: The Unstoppable Force (`autovacuum_freeze_max_age`)

What happens if a database is inactive (e.g., `template0`), or if autovacuum is disabled? Old unfrozen tuples could lead to wraparound.



The Final Safeguard: `autovacuum_freeze_max_age`

- **Default Value:** 200,000,000 transactions. (Max allowed is ~2 billion).

How it Works

- The autovacuum launcher constantly monitors the age of `datfrozenxid` for every database.
- If any database's `datfrozenxid` age threatens to exceed `autovacuum_freeze_max_age`, PostgreSQL will forcibly start a special autovacuum process on that database.

Key Features of this Forced Vacuum

- It **cannot be cancelled** by the user.
- It is triggered even if `autovacuum` is turned off for the system or the specific tables.

Side Benefit

- This process is also responsible for truncating the Commit Log (CLOG), as transaction statuses older than the oldest `datfrozenxid` in the cluster are no longer needed.

NotebookLM

Layer 4: Failsafe Mode (vacuum_failsafe_age)

In the most critical situations, when a wraparound event is imminent, a final emergency mode is activated.



The Trigger: `vacuum_failsafe_age`

- **Default Value:** 1,600,000,000 (1.6 billion) transactions.
- This is typically set to be a large percentage of `autovacuum_freeze_max_age`.

Emergency Measures

When the age of unfrozen transactions approaches this threshold, the forced autovacuum process goes into 'failsafe' mode. To freeze tuples as fast as possible, it takes drastic measures:

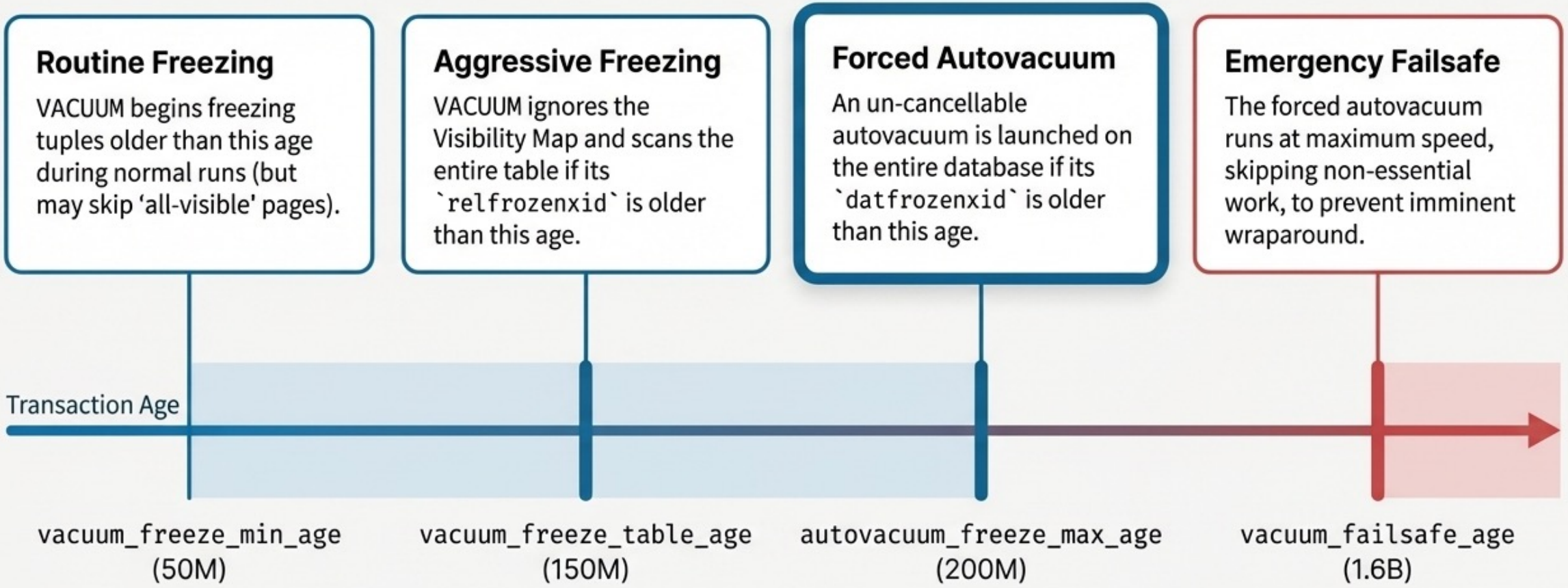
Ignores Cost Delays: It runs at full speed, disregarding `autovacuum_vacuum_cost_delay`.

Skips Index Cleanup: Index vacuuming is skipped to focus all I/O on scanning heap pages and freezing tuples.

This is the system's last-ditch effort to prevent a forced shutdown due to impending data corruption.

NotebookLM

PostgreSQL's Four Layers of Wraparound Defense



NotebookLM

Taking Manual Control of Freezing

While autovacuum manages freezing automatically, there are times when manual intervention is necessary or more efficient.

1. Forcing a Full Table Freeze: VACUUM (FREEZE)

- This command performs an aggressive freeze on a table.
- It behaves as if `vacuum_freeze_min_age` were set to `0`, attempting to freeze all eligible tuples, regardless of age.
- For maximum speed in an emergency, you can also disable index cleanup:

```
VACUUM (FREEZE, INDEX_CLEANUP FALSE);
```

2. Freezing Data on Initial Load: COPY ... WITH (FREEZE)

- Extremely useful for loading static or archival data.
- Tuples are frozen as they are loaded into the table.
- This sets the 'all-visible' and 'all-frozen' flags on pages immediately, meaning VACUUM won't need to touch the table again unless the data is modified.

```
COPY my_static_table FROM stdin WITH (FREEZE);
```

Limitation: Requires an exclusive lock and is only possible if the table is created or truncated in the same transaction.

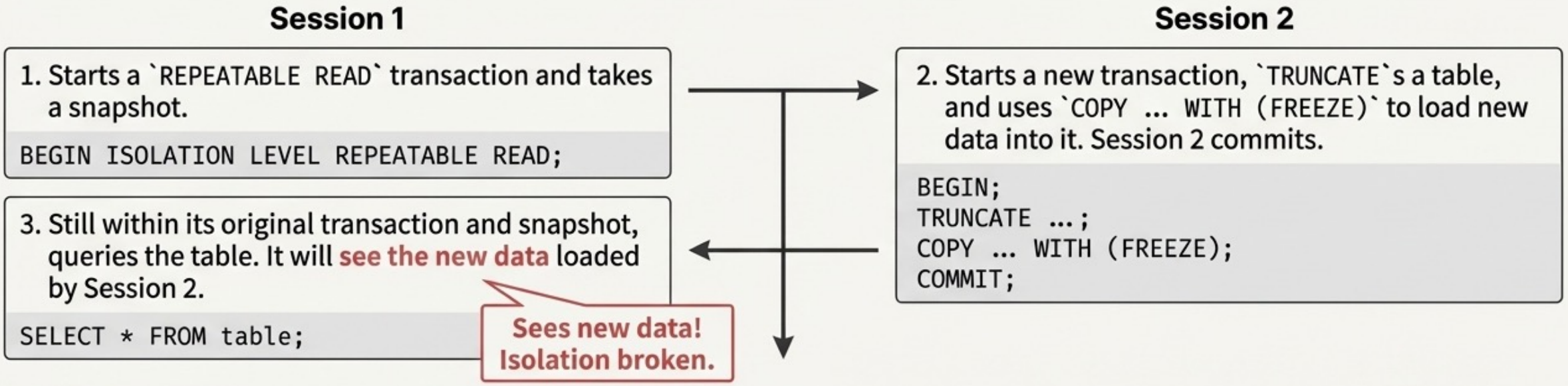
NotebookLM

The `COPY FREEZE` Caveat: A Break in Isolation

The `COPY ... WITH (FREEZE)` command is powerful, but it comes with a significant side effect related to transaction isolation.

The Anomaly

Because frozen tuples are immediately visible to *all* snapshots, they can violate the `REPEATABLE READ` isolation level.



Why it Happens

The exclusive lock on the table prevents concurrent access *during* the load, but the "frozen" status of the new tuples bypasses normal snapshot visibility rules once the lock is released.

This is a documented and deliberate trade-off, but it's critical to be aware of when performing bulk data loads on a live system.

NotebookLM

8. Rebuilding Tables and Indexes



Taming PostgreSQL Bloat

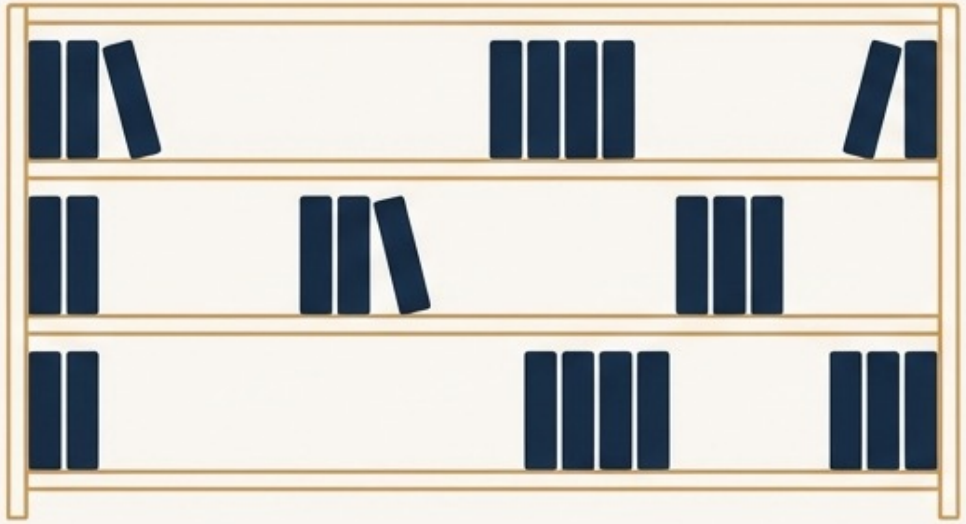
A Strategic Guide to Table and Index Reorganization for Optimal Performance

NotebookLM

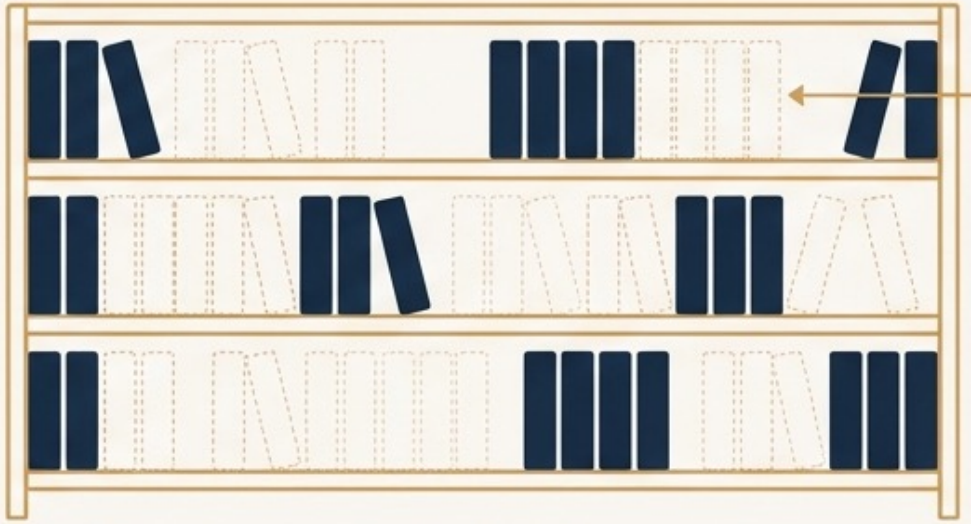
Why isn't your regular `VACUUM` shrinking your database?

Standard `VACUUM` is essential for routine maintenance, but it has a key limitation. It reclaims space *within* data pages for future writes, but it rarely reduces the total number of pages or shrinks the file on disk.

Before



After `VACUUM`



Space is reusable, but the file size remains.

To reclaim disk space, you need more than just `VACUUM`.

NotebookLM

The compounding cost of ‘bloat’ on performance.



Slower Scans: Full table and index scans take longer as they must read through more, mostly empty, pages.



Inefficient Caching: The buffer cache is filled with pages that have low data density, wasting valuable memory.



Deeper B-Trees: Bloated indexes can gain extra levels, slowing down index lookups.



Increased Storage & Backup Costs: Physical file sizes are unnecessarily large, consuming more disk space and increasing backup times.

NotebookLM

🔬 Measuring data density with `pgstattuple`.

The `pgstattuple` extension provides a ground-truth analysis of space utilization within your tables and indexes.

Setup

```
-- First, enable the extension in your database
CREATE EXTENSION pgstattuple;
```

Baseline Example

```
-- A table freshly loaded with 500,000 rows
=> SELECT * FROM pgstattuple('vac') \gx

-[ RECORD 1 ]----+------------
table_len        | 70623232
tuple_count      | 500000
tuple_len        | 64500000
tuple_percent    | 91.33
dead_tuple_count | 0
free_space       | 381844
free_percent     | 0.54
```



High percentage indicates healthy, dense data storage.

Note For indexes, the key metric is `avg_leaf_density` from the `pgstatindex()` function.

NotebookLM

Creating bloat and proving `VACUUM's limits.

Step 1: Create Bloat

We delete 90% of the rows from our sample table.

```
-- Original size: Table 67 MB, Index 109 MB
DELETE FROM vac WHERE id % 10 != 0;
-- DELETE 450000
```

Step 2: Run Standard Vacuum

We run a standard `VACUUM` to clean up dead tuples.

```
VACUUM vac;
```

Step 3: Check The Size

The 'before and after' table shows the file sizes are unchanged.

Object	Size Before `DELETE`	Size After `DELETE` & `VACUUM`
Table 'vac'	67 MB	**67 MB**
Index 'vac_s'	109 MB	**109 MB**

Conclusion: `VACUUM` cleaned up the dead tuples, but the OS hasn't reclaimed any space because there were no empty pages at the end of the file.

NotebookLM

The evidence of bloat is in the density, not the file size.

We run `pgstattuple` and `pgstatindex` again on our bloated table.

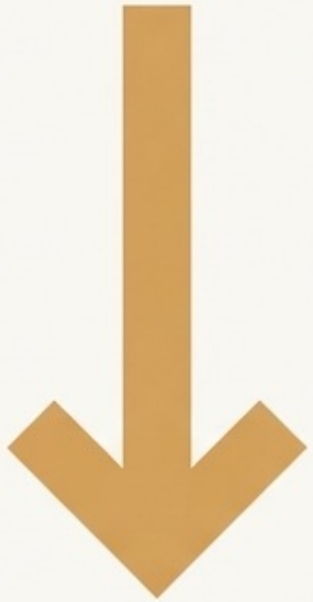
Before (Healthy State)

Table Density (`tuple_percent`)

91.3%

Index Density (`avg_leaf_density`)

53.8%



After (Bloated State)

Table Density (`tuple_percent`)

9.13%

Index Density (`avg_leaf_density`)

6.71%

Key Insight: The files are still large, but now they are mostly empty space. This is the bloat we need to fix.

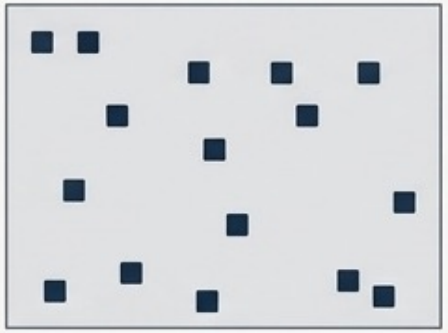
NotebookLM

🔧 The direct approach: Rebuilding with `VACUUM FULL`.

`VACUUM FULL` is a command that aggressively reclaims space by completely rewriting the table and all its indexes into new disk files.

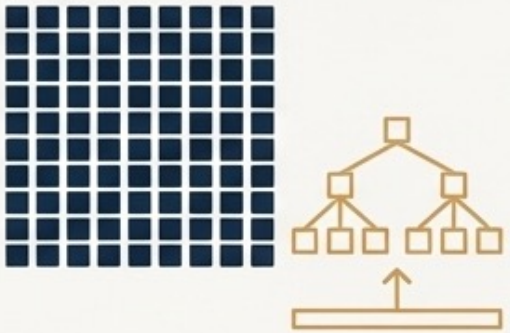
How it Works

1.



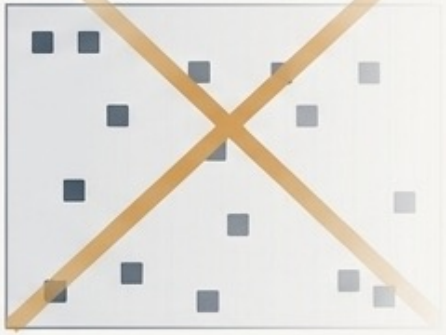
A new, tightly packed copy of the table is created.

2.



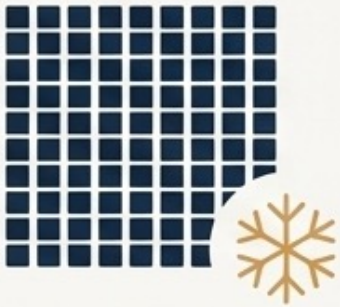
Each index on the table is completely rebuilt from scratch.

3.



The original, bloated data files are removed.

4.

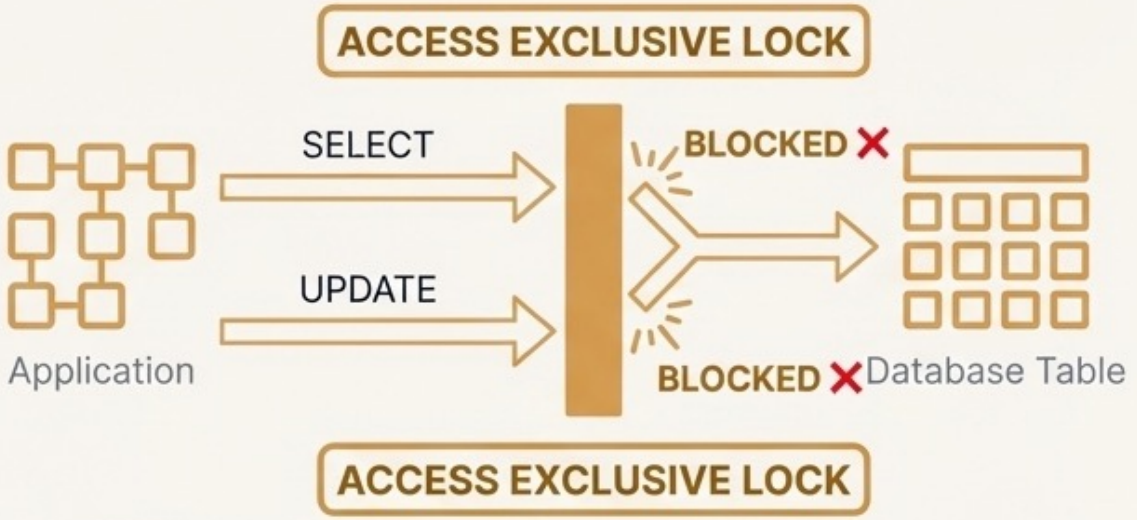


Tuples are 'frozen' as part of the process, which helps with transaction ID wraparound.

Side Note: The progress of the operation can be monitored using the `pg_stat_progress_cluster` view.

NotebookLM

The high cost of `VACUUM FULL`: An exclusive lock.



The Problem

During its entire execution, `VACUUM FULL` acquires an `ACCESS EXCLUSIVE` lock on the table.

What this means for you

- **ALL reads are blocked.** (No `SELECT` queries)
- **ALL writes are blocked.** (No `INSERT`, `UPDATE`, `DELETE`)
- The table is completely inaccessible to your application for the duration of the command.

Additional Consideration: The operation requires extra disk space, as the old and new files must exist simultaneously during the rebuild.

NotebookLM

The result: A dramatically smaller and denser table.

Metric	Before `VACUUM FULL`	After `VACUUM FULL`
Table Size (`vac`)	67 MB	**6904 kB**
Index Size (`vac_s`)	109 MB	**6504 kB**
Table Density (`tuple_percent`)	9.13%	**91.23%**
Index Density (`avg_leaf_density`)	6.71%	**91.08%**
Table File Path	`base/16391/16514`	`base/16391/16526`



Key Insight: Rebuilding an index from scratch (`VACUUM FULL`) often results in **even better density** than the original, sequentially-loaded index.

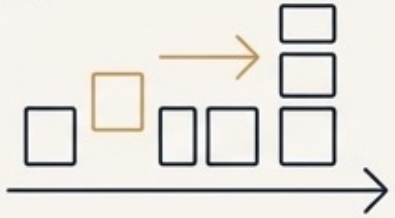
NotebookLM

Exploring related native rebuilding commands

CLUSTER

Functionally identical to `VACUUM FULL`, but with one key difference: it physically re-sorts tuples on disk according to a specified index.

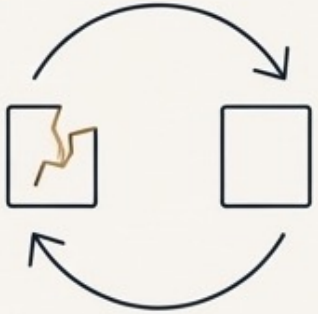
- Benefit: Can significantly improve performance for queries that use index scans on the chosen index.
- Caveat: The physical order is not maintained after subsequent writes.



REINDEX

Rebuilds one or more specified indexes without rewriting the table itself.

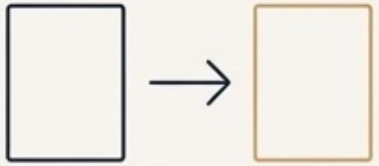
- Useful for **fixing** bloated or corrupted indexes specifically. `VACUUM FULL` and `CLUSTER` use this command internally.



TRUNCATE

Logically equivalent to a `DELETE` without a `WHERE` clause, but much faster.

- It immediately removes **all rows by creating a** new, empty file, bypassing the `VACUUM` process entirely.

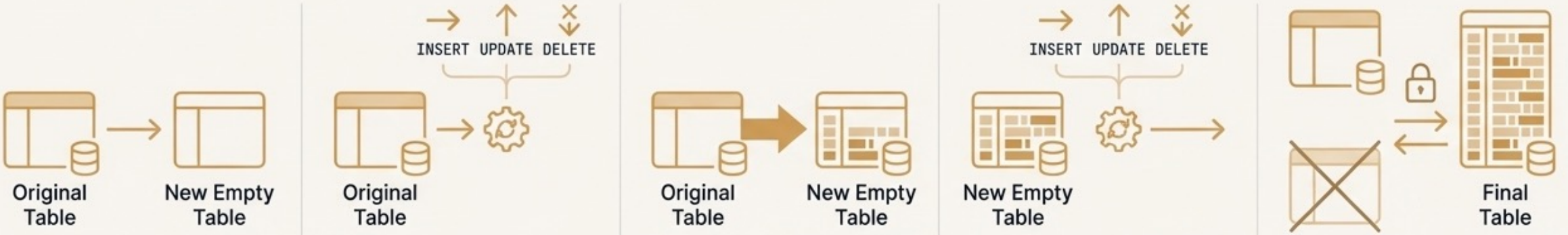


NotebookLM

The production solution: Online reorganization with `pg_repack`.

The Challenge: How to achieve the benefits of `VACUUM FULL` without the downtime of an exclusive lock?

The `pg_repack` Strategy



1. Creates a new, empty copy of the target table.
2. Installs triggers on the original table to capture all ongoing changes (`INSERT`, `UPDATE`, `DELETE`).
3. Copies all existing data from the original table to the new one.
4. Applies all the captured changes to the new table to bring it up to date.
5. Acquires a brief exclusive lock to swap the table files in the system catalog, then drops the old table.

🔍 **Key Benefit:** The disruptive exclusive lock is held for only a very short time at the very end of the process, making it suitable for high-availability systems.

🌱 **Alternative Tool:** Briefly mention `pgcompacttable` as another approach that works in-place by performing 'fake' updates to gradually move tuples, requiring no extra disk space but taking longer.

NotebookLM

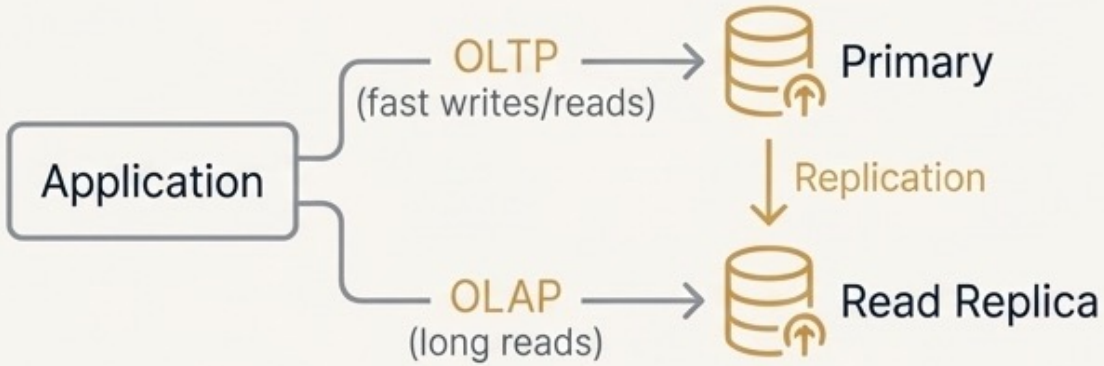
Preventing bloat: Taming long-running transactions

The Cause

A long-running transaction can establish a "horizon" that prevents `VACUUM` from cleaning up dead tuples, even if those tuples were modified by other, completed transactions.

Architectural Solution

Segregate workloads. Keep fast OLTP queries on the primary server. Route long-running, analytical OLAP queries to a read replica. This isolates their impact from the primary's write workload.



Configuration Safety Nets

For misbehaving applications or drivers, use these parameters as a last resort:

- `idle_in_transaction_session_timeout`
  Automatically terminates sessions that are idle within a transaction for too long.
- `old_snapshot_threshold`
  Allows `VACUUM` to clean up old tuples even if a long-running transaction needs them, which will cause an error for that transaction ("snapshot too old") but protects the health of the database.

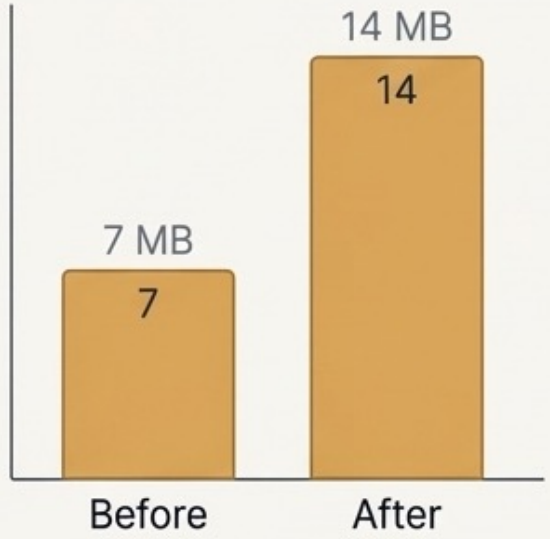
NotebookLM

💡 Pro-Tip: Avoid bloat from mass updates with batching.

The Problem

A single transaction that updates every row in a table will effectively double the table size, as all the old tuple versions must be retained until the transaction commits.

```
-- A single UPDATE doubles the table size from 7 MB to 14 MB
UPDATE vac SET processed = true;
```



The Solution

Process changes in smaller, independent batches. Between batches, `autovacuum` can run and reclaim space, allowing new tuple versions to fill the freed space.

The Pattern: `FOR UPDATE SKIP LOCKED`

```
-- Process 1000 rows at a time in a loop
WITH batch AS (
    SELECT id FROM vac
    WHERE NOT processed
    LIMIT 1000
    FOR UPDATE SKIP LOCKED
)
UPDATE vac SET processed = true
WHERE id IN (SELECT id FROM batch);
```

`SKIP LOCKED` tells PostgreSQL to ignore rows already locked by other concurrent batch workers, making this a robust and scalable pattern.

NotebookLM

The PostgreSQL Bloat Management Playbook.



DIAGNOSE

Tool: `pgstattuple` & `pgstatindex` extensions.

Key Metric: Look for low `tuple_percent` in tables and low `avg_leaf_density` in indexes.

Trigger: Investigate when density falls below an acceptable threshold (e.g., 70%).



REMEDIATE

For Maintenance Windows: Use `VACUUM FULL` or `CLUSTER` for the most direct and efficient rebuild.

For High-Availability Systems: Use the `pg_repack` extension to perform an online reorganization with minimal locking.

For Indexes Only: Use `REINDEX` to fix index-specific bloat.



PREVENT

Against Mass Updates: Break large `UPDATE/DELETE` jobs into smaller batches using the `FOR UPDATE SKIP LOCKED` pattern.

Against Long Transactions: Isolate analytical queries on read replicas. Use `idle_in_transaction_session_timeout` as a safeguard.

NotebookLM



Proactive management is the key to sustained performance.

Understanding and controlling bloat is not a one-time fix. It's an essential discipline for building scalable, resilient, and performant PostgreSQL systems.

By diagnosing, remediating, and—most importantly—preventing bloat, you move from reactive firefighting to proactive database architecture.

NotebookLM

9. Buffer Cache

The Heart of PostgreSQL: A Deep Dive into the Buffer Cache

Understanding How PostgreSQL Manages Memory for Peak Performance

Dive into the Buffer Cache

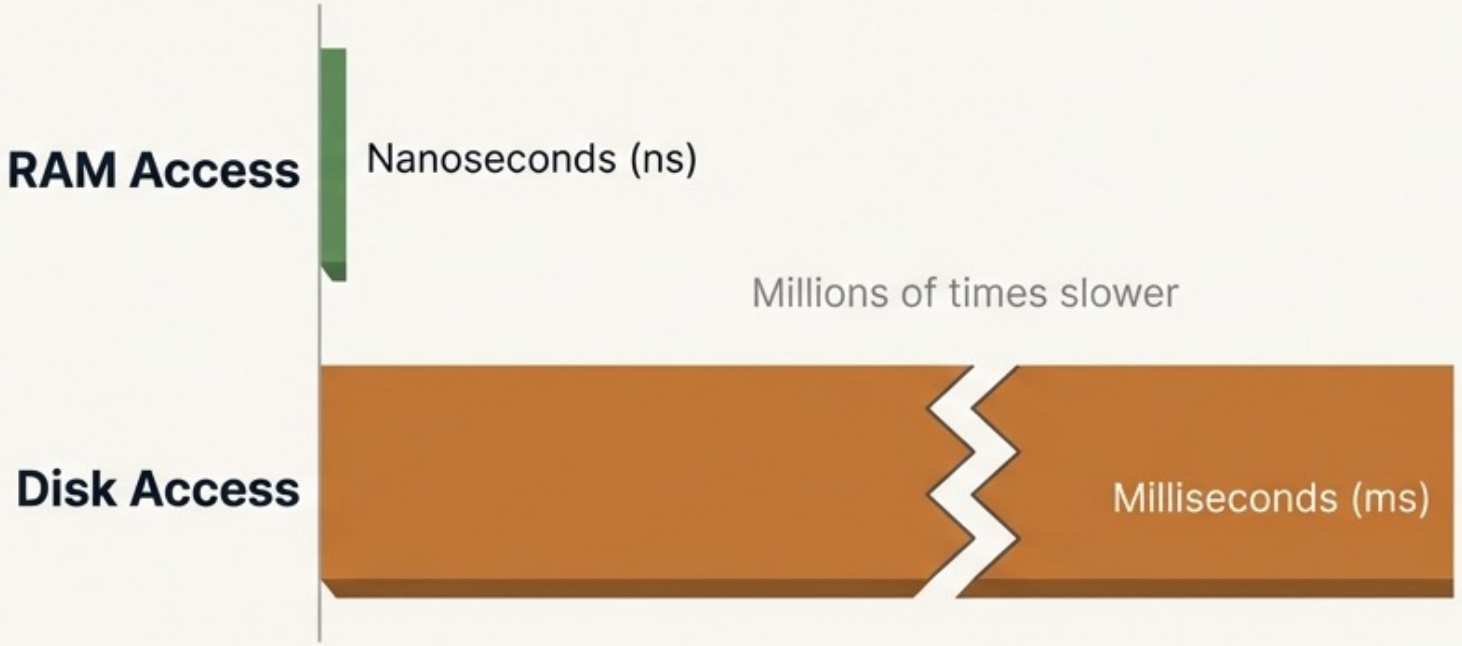
Understanding How PostgreSQL Manages Memory for Peak Performance



NotebookLM

The Problem: A Chasm Between Memory and Disk

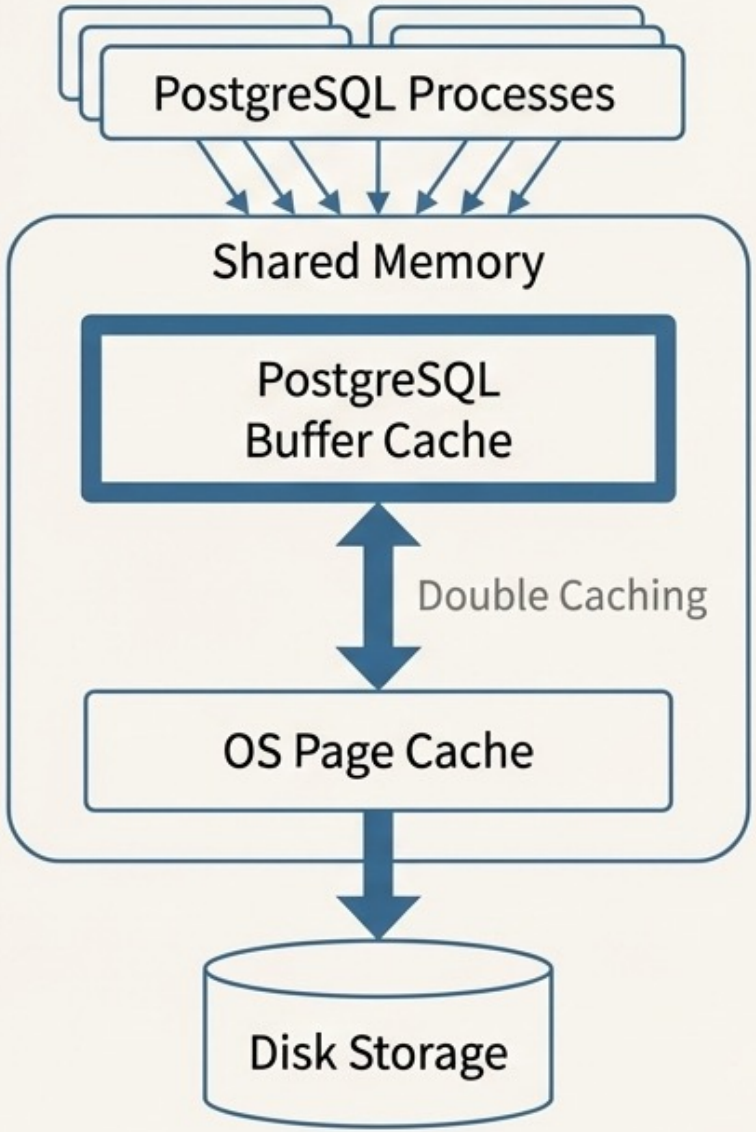
The primary goal of caching is to bridge the performance gap between fast and slow memory. PostgreSQL's buffer cache holds relation pages to balance the vast difference in access times between disk and RAM.



NotebookLM

The Solution: A Shared Buffer Cache

The buffer cache is a large, critical data structure located in the server's shared memory, making it accessible to all PostgreSQL processes.



PostgreSQL's Approach

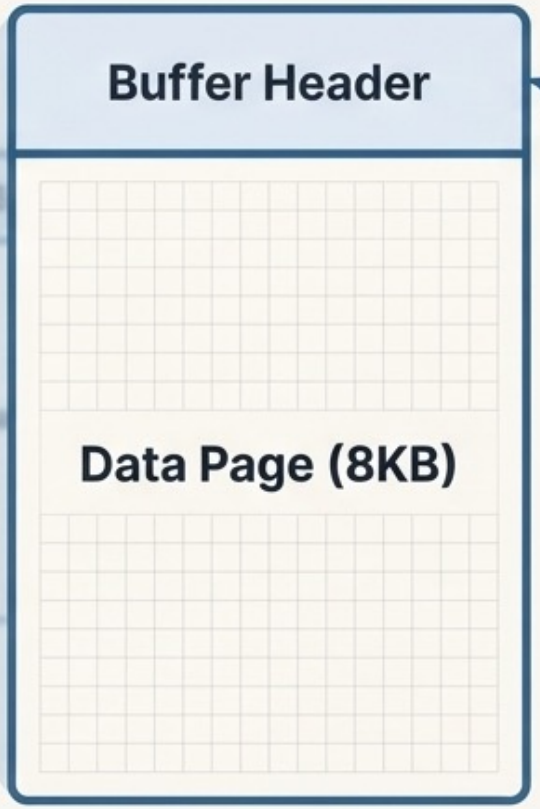
- Unlike many databases that use Direct I/O to bypass the OS cache, PostgreSQL reads and writes through the OS's own file cache. This is known as "double caching."
- While this can introduce overhead, the community is actively working on Direct and Asynchronous I/O support. Direct I/O would allow PostgreSQL to access memory directly (DMA), but requires implementing features like data prefetching separately.

NotebookLM

The Anatomy of the Cache: Buffers

The buffer cache is structured as a simple array of fixed-size buffers. Each buffer is a chunk of memory that holds two distinct parts: the actual data page and a descriptive header.





Physical Location: The page's unique identity (File ID, Fork, Block Number).

Dirty Flag: A boolean indicating if the page has been modified in memory and needs to be written back to disk.

Usage Count: A counter tracking how frequently the buffer has been accessed.

Pin Count (Reference Count): A counter tracking how many processes are actively using the buffer.

NotebookLM

The Problem: Finding a Needle in a Haystack

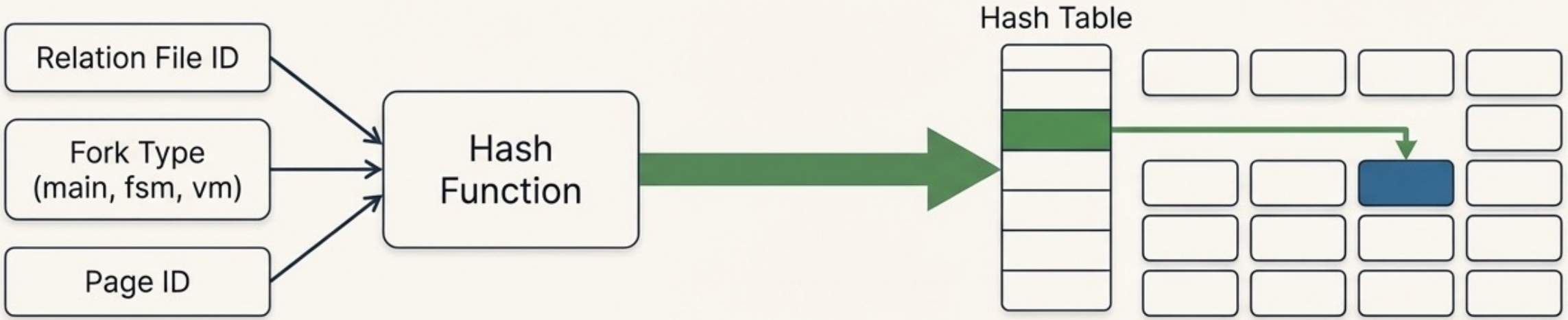
When a process needs a specific data page, it requests it from the buffer manager. Searching the entire array of buffers linearly for that page would be catastrophically slow. The system needs a lookup mechanism with near-constant time (O(1)) performance.



NotebookLM

The Solution: A Hash Table for Instant Lookups

PostgreSQL uses a hash table in shared memory to map a **page's identity** to a specific buffer ID. When a page is requested, the system computes a hash key to find the corresponding buffer instantly. This is a "cache hit."

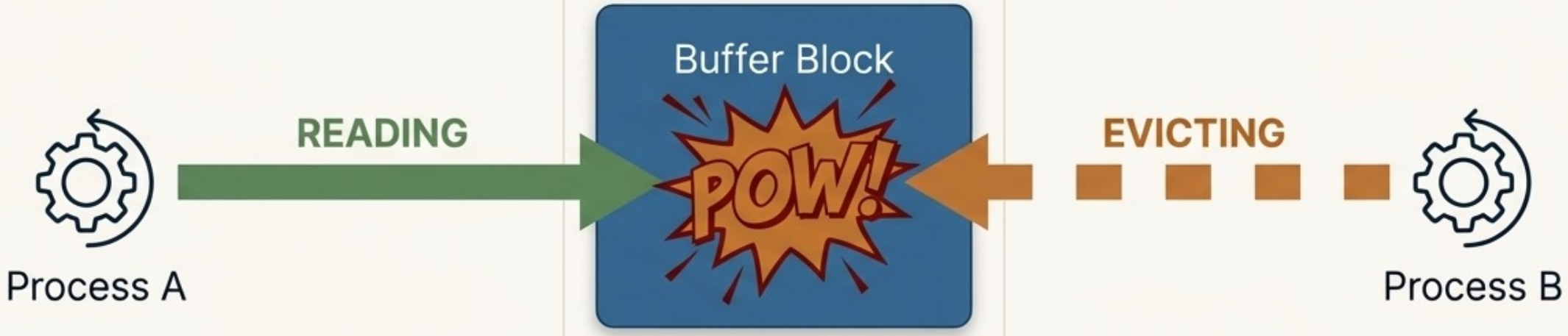


⚠ This hash-based design has been criticized because it isn't efficient for finding all buffers belonging to a specific table, which is needed for commands like `DROP` or `TRUNCATE`.

NotebookLM

The Problem: How to Prevent Chaos During Concurrent Access

Multiple backend processes can access the buffer cache simultaneously. If a process needs a buffer for a new page, it might evict a page that another process is actively reading. This would lead to data corruption or crashes. The system requires a mechanism to “lock” a buffer while it’s in use.



NotebookLM

The Solution: “Pinning” Buffers

When a process accesses a buffer, it “pins” it by incrementing the `pin count` in the buffer’s header. A buffer with a `pin count` greater than 0 is considered **in-use** and cannot be evicted by the replacement algorithm.

In Practice: Observing Pins

- `EXPLAIN (buffers)`: The output `shared hit=1` confirms a page was found in the cache and pinned during the query.
- `pg_buffercache`: An open cursor will hold a pin on a buffer, visible in the view as `pins = 1`.
- `VACUUM`: A verbose vacuum log may show the message Skipped 1 page due to buffer pins, as it cannot clean tuples from a page currently in use by another transaction.

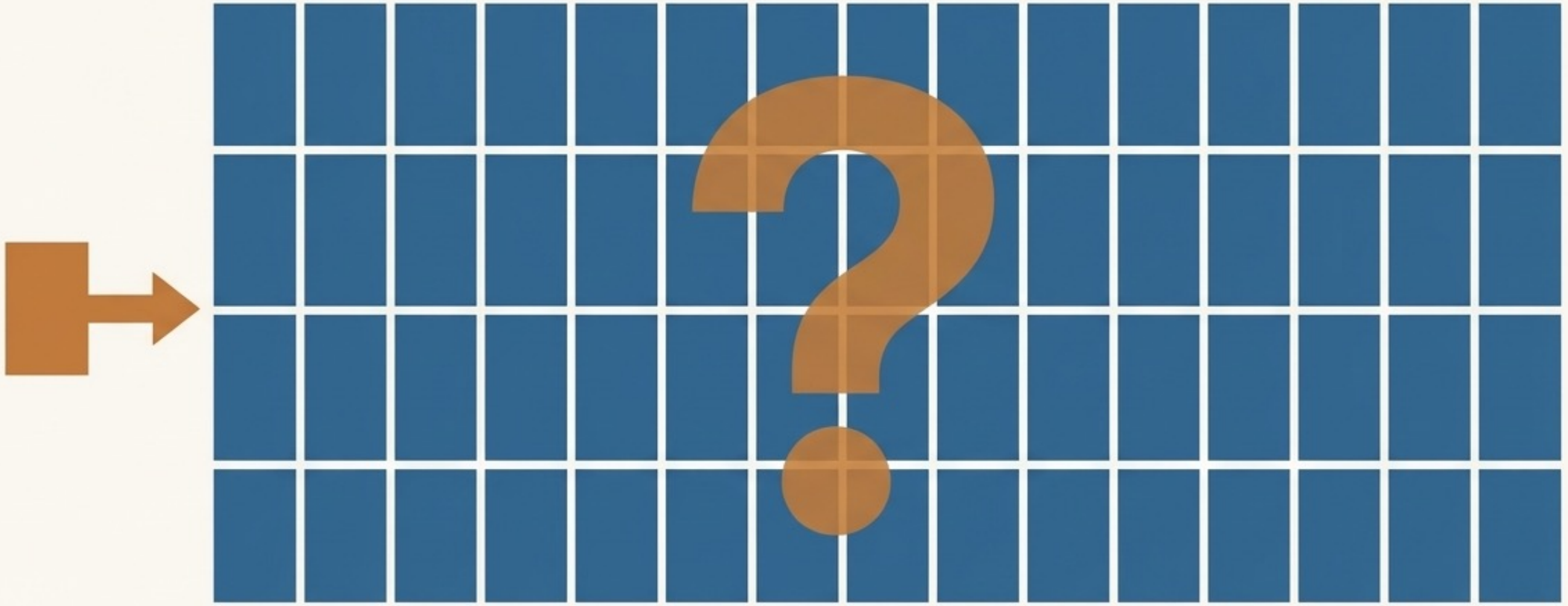
```
-- During an open cursor transaction                    #888888
SELECT * FROM buffercache('cacheme');

bufferid | relfork | relblk | isdirty | usagecount | pins
---------+---------+--------+---------+------------+-----
     268 | main    |      0 | t       |          3 |    1
```

NotebookLM

The Problem: The Cache is Full

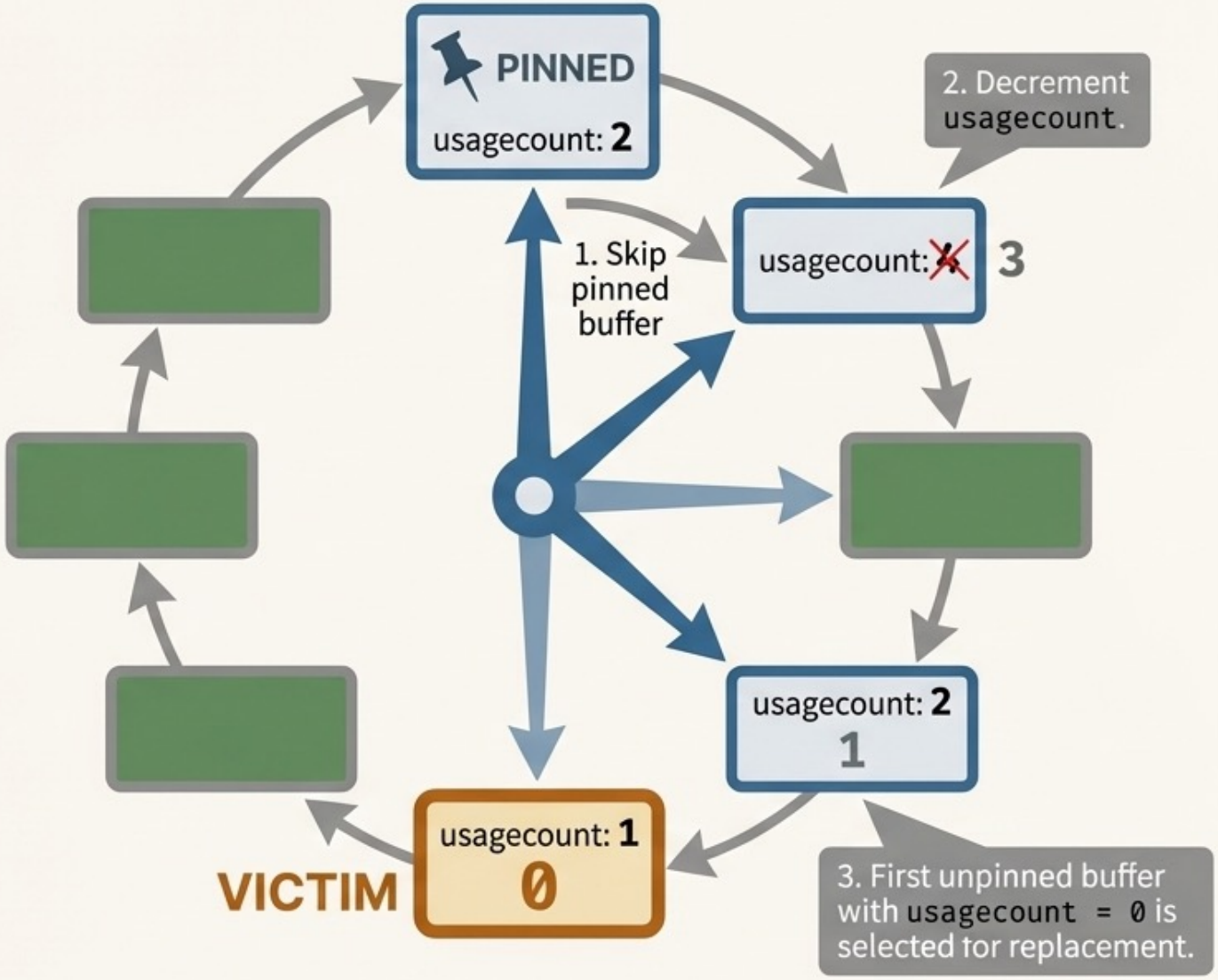
Eventually, all buffers in the cache will be occupied. When a page is requested that is not in the cache (a **cache miss**), the buffer manager must select a victim buffer to evict. The choice is critical for performance: evicting a “hot,” frequently-used page will cause performance degradation, while evicting a “cold” page will have minimal impact. The goal is to always evict the least useful page.



NotebookLM

The Solution: The Clock Sweep Replacement Algorithm

PostgreSQL uses a “Clock Sweep” algorithm, which approximates a Least Recently Used (LRU) policy. A conceptual “clock hand” sweeps through the array of buffers.



Key Detail: The `usagecount` is capped at 5. This prevents very ‘hot’ pages from requiring many full sweeps of the clock before becoming eligible for eviction.

NotebookLM

The Problem: The “Tsunami Scan”

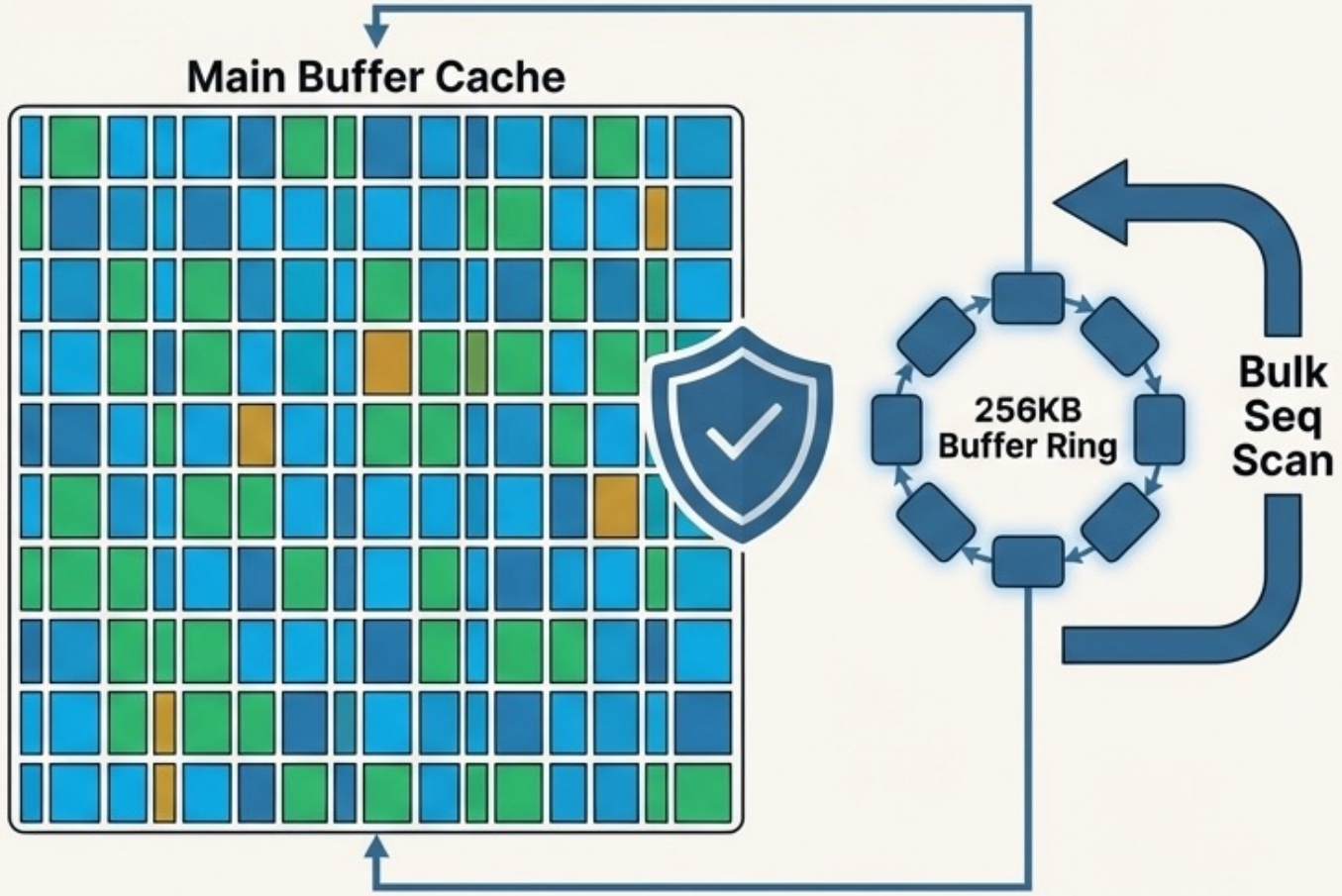
A single, massive query—like a sequential scan on a very large table—can be disastrous for cache performance. The query reads thousands of pages, each one potentially evicting a valuable, “hot” page from the cache. By the time the scan is done, the entire cache is filled with its single-use data, and all the frequently accessed index and data pages are gone. This is called cache pollution.



NotebookLM

The Solution: Containing Bulk Operations with Buffer Rings

To prevent cache pollution, PostgreSQL uses a small, dedicated 'ring' of buffers for bulk operations. The eviction strategy is confined entirely within this ring, protecting the vast majority of the buffer cache.



- **Bulk Read:** For large sequential scans. Uses a small 256KB ring.
- **Bulk Write:** For `COPY FROM`, `CREATE TABLE AS`. Uses a larger 16MB ring.
- **Vacuum:** For full table scans. Uses a 256KB ring.

Important Caveat: Buffer rings are not used for all large operations. Index scans, large `UPDATE`/`DELETE`s, and `TOAST` table access do not use this strategy and can still pollute the cache.

NotebookLM

The Administrator's Toolkit: Sizing and Monitoring

Sizing (`shared_buffers`)

The default of 128MB is too low for production. A common recommendation is to start with 25% of available system RAM. However, there is no magic number. The best approach is to experiment with different sizes and monitor the performance of a realistic workload.

Monitoring (`pg_buffercache`)

This extension is the primary tool for introspection. It allows you to see the state of every buffer in the cache.

In Practice: A Powerful Monitoring Query

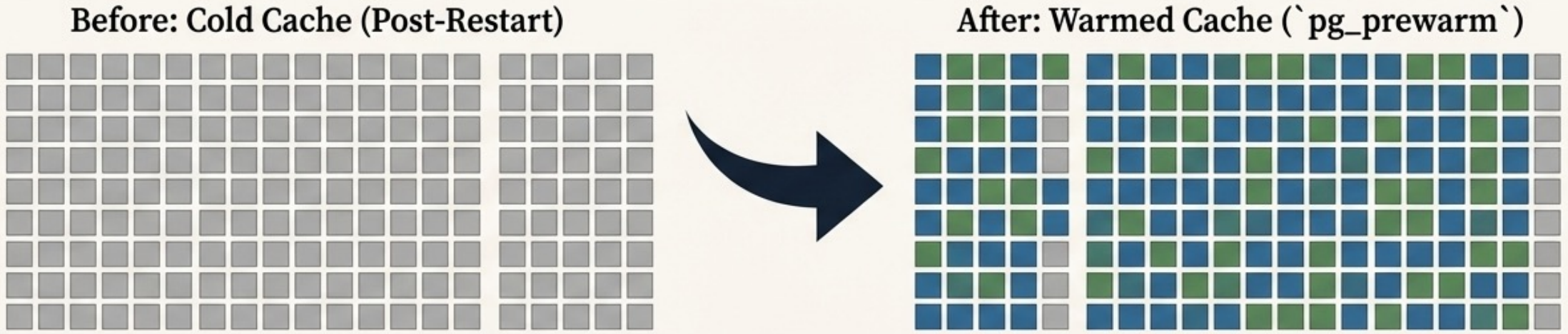
This query shows how much of each relation is currently in the cache and, more importantly, what percentage of it is 'hot' (frequently used, with a `usagecount > 1`).

```
SELECT
  c.relname,
  count(*) AS blocks_in_cache,
  round(100.0 * 8192 * count(*) / pg_table_size(c.oid)) AS "%
of rel",
  round(100.0 * 8192 * count(*) FILTER(WHERE b.usagecount > 1) /
    pg_table_size(c.oid)) AS "% hot"
FROM pg_buffercache b JOIN pg_class c ON ...
GROUP BY c.relname, c.oid ORDER BY 2 DESC LIMIT 10;
```

NotebookLM

The Administrator's Toolkit: Solving the Post-Restart Slump

After a server restart, the buffer cache is empty, or "cold." The initial queries will be slow as they must read data from disk. This initial period of poor performance can be solved by "warming" the cache.



The Solution: `pg_prewarm` extension

- Allows you to pre-load specific, critical relations into the buffer cache on demand.
- **Autoprewarm Feature**: When enabled, an `autoprewarm leader` process periodically saves a list of the pages currently in the cache to **`PGDATA/autoprewarm.blocks`**. After a restart, it automatically loads those pages back into the cache, significantly reducing warm-up time.

NotebookLM

One Exception and Your Key Takeaways

The Exception: Local Caches for Temporary Tables

Temporary tables are private to a single session and do not use the shared buffer cache. Instead, each backend uses a simpler, private local buffer cache for its temporary data, controlled by `temp_buffers`.

1. 

The Engine

The buffer cache is PostgreSQL's primary I/O performance engine, bridging the immense speed gap between RAM and disk.

2. 

The Mechanisms

Lookups are instant via a hash table. Concurrency is safely managed by pinning.

3. 

The Intelligence

A 'Clock Sweep' algorithm intelligently evicts the least-used pages, approximating LRU.

4. 

The Protection

Buffer rings protect the cache from pollution during large sequential scans and bulk loads.

5. 

Your Job

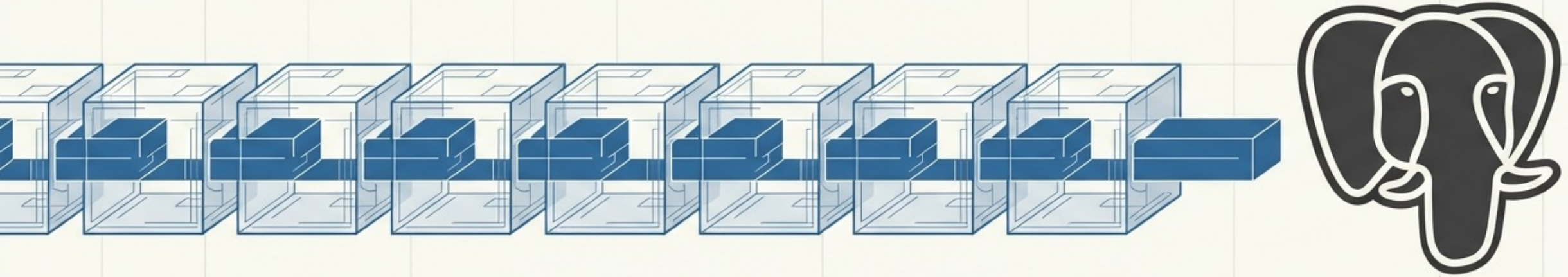
Size it wisely (shared_buffers), monitor it (pg_buffercache), and warm it after restarts (pg_prewarm).

NotebookLM

10. Write-Ahead Log

PostgreSQL Internals: A Deep Dive into the Write-Ahead Log

Understanding Durability, Recovery, and Performance



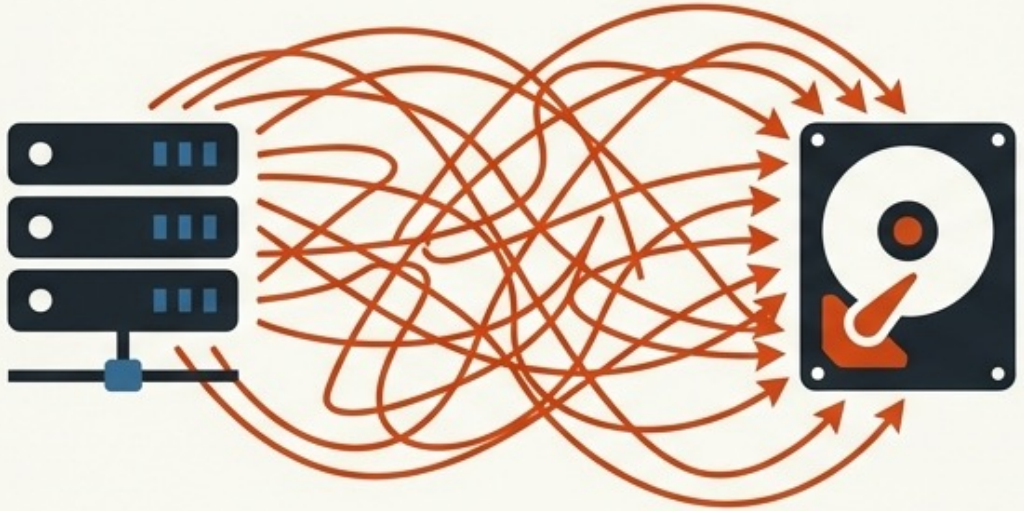
This presentation deconstructs the Write-Ahead Log (WAL), the mechanism that underpins the ‘D’ for Durability in ACID transactions. We will explore its architecture, its role in crash recovery, and how to tune it for optimal performance.

NotebookLM

The Durability Dilemma: How to Guarantee Safety Without Sacrificing Speed

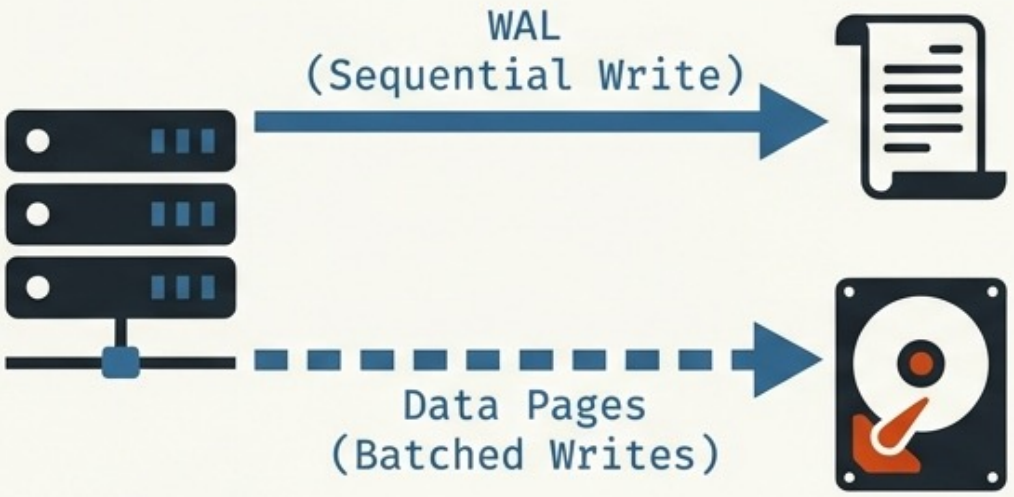
The Core Problem: In a crash (power loss, OS error), all data in RAM is lost. We must ensure that no committed changes are lost, restoring the database to a consistent state upon restart.

The Naive Approach: Slow & Inefficient



Result: Massive, slow, random disk I/O, crippling performance.

The PostgreSQL Way: Fast & Durable



Solution: Separate the *logging* of a change from the *writing* of the data itself.

NotebookLM

The WAL Principle: First, Log the Change. Then, Change the Data.

The Write-Ahead Log (WAL) is a continuous stream of records describing every critical change made to the database.

The Golden Rule

A log record describing a page modification ***must*** be written to durable storage ***before*** the modified data page itself is written to disk.

What Gets Logged

- ✓ Page modifications in the buffer cache.
- ✓ Transaction commits and rollbacks (changes to CLOG).
- ✓ File system operations (e.g., creating/deleting tables).

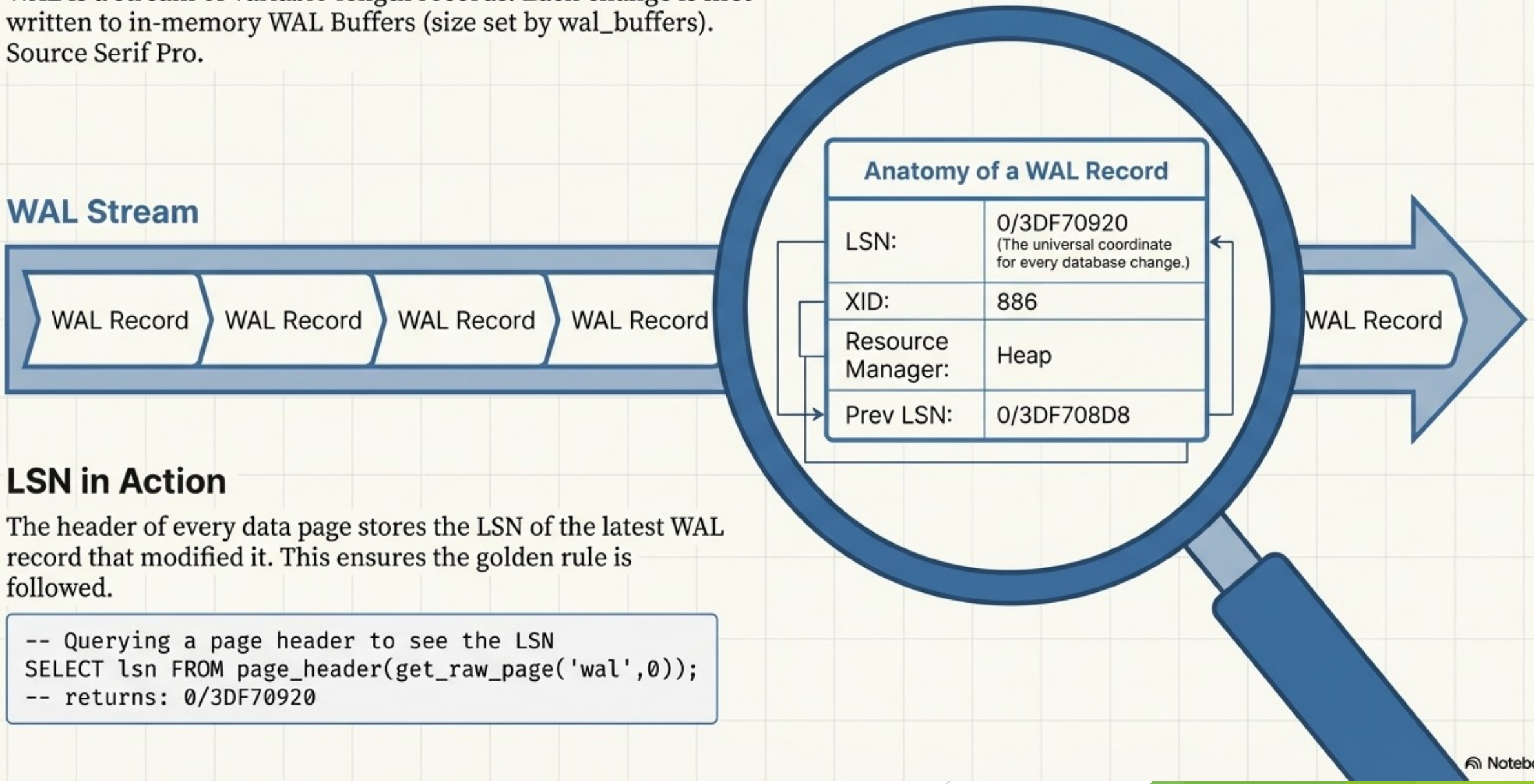
What Does Not Get Logged

- ✗ Operations on UNLOGGED tables.
- ✗ Operations on temporary tables.

NotebookLM

The Logical Structure: An Ordered Stream of Change Records

WAL is a stream of variable-length records. Each change is first written to in-memory WAL Buffers (size set by wal_buffers). Source Serif Pro.



LSN in Action

The header of every data page stores the LSN of the latest WAL record that modified it. This ensures the golden rule is followed.

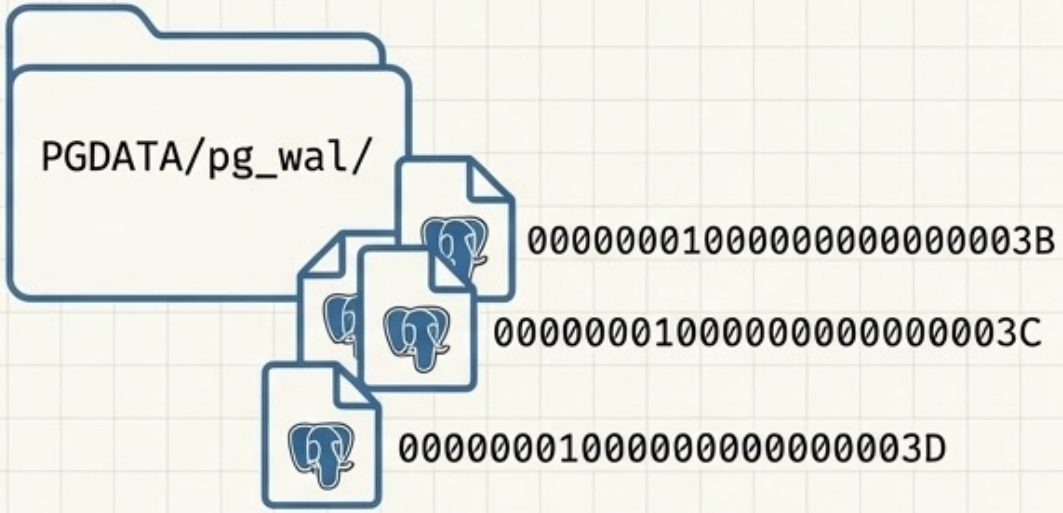
```
-- Querying a page header to see the LSN
SELECT lsn FROM page_header(get_raw_page('wal',0));
-- returns: 0/3DF70920
```

NotebookLM

The Physical Structure: Storing the Stream in Segments

Location

The logical stream is broken into physical files called **segments**, stored in the `PGDATA/pg_wal/` directory.



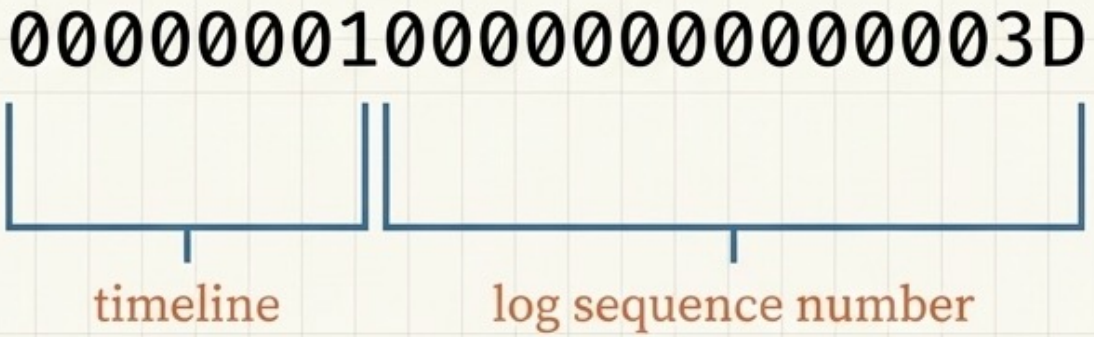
Segment Files

- Typically 16MB each (configurable at initdb with `--wal-segsize`).
- New records are appended to the current segment until it's full, then a new segment is created.

Inspection Tool

The `pg_waldump` utility allows you to read the contents of WAL segment files directly from the operating system.

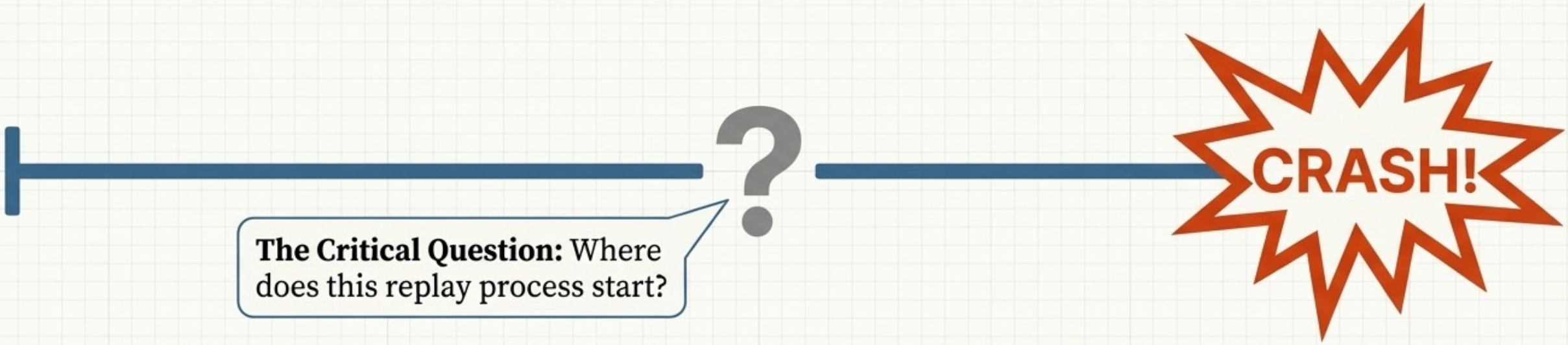
Decoding File Names



NotebookLM

The Problem of the Infinite Log

The Challenge: To recover from a crash, PostgreSQL must replay the WAL to apply any changes that didn't make it from RAM to the main data files.



Start too late?
→ You miss changes and end up with with corrupted data.

Start from the very beginning?
→ Requires infinite storage and could lead to recovery taking days.

The Solution: We need a mechanism to periodically establish a safe "starting line" for recovery. This mechanism is the **Checkpoint.**

NotebookLM

Checkpoints: Creating a Safe Starting Line for Recovery

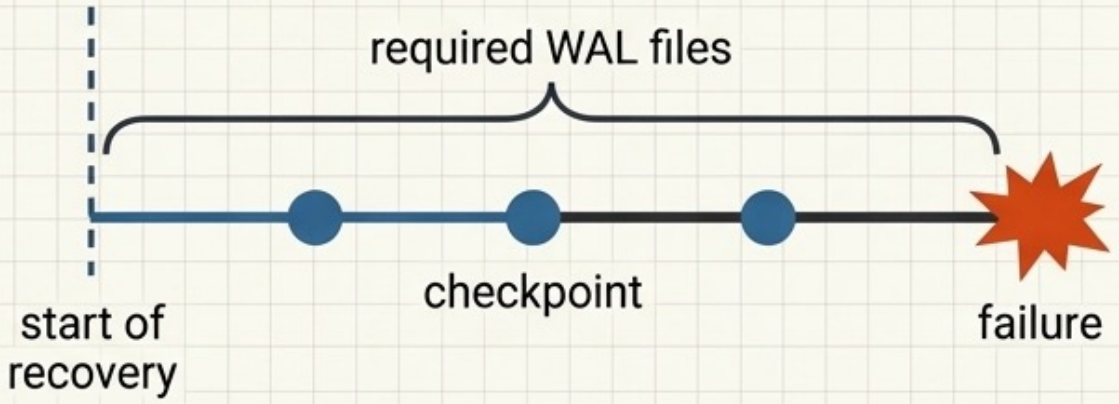
Definition:
A checkpoint is a point in the WAL sequence *before which* all data file modifications have been flushed from memory (dirty buffers) to disk.

The Outcome:
This process establishes a **REDO point,** a specific LSN from which PostgreSQL knows it is safe to begin recovery after a crash.

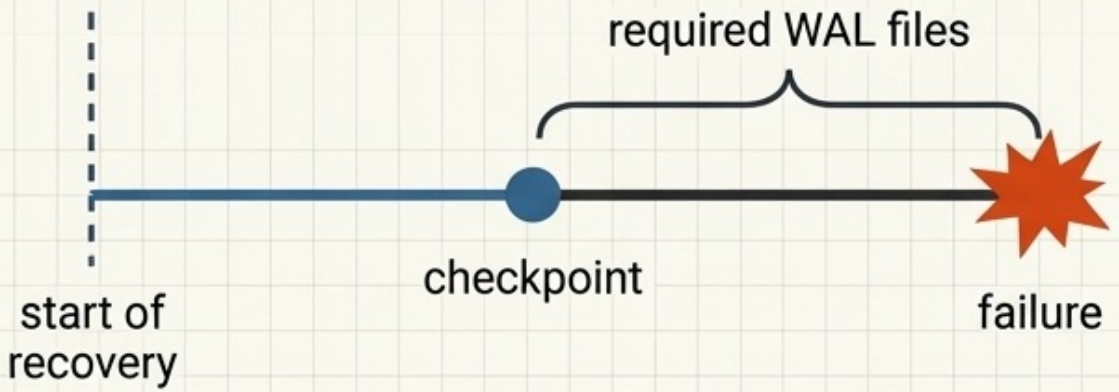
The Benefit:
Once a checkpoint is complete, all WAL segments older than the new REDO point can be safely removed or recycled.

The 'Fuzzy' Process:
This is not a 'stop-the-world' event. A dedicated background **checkpointer** process spreads the work over time to minimize I/O impact.

Before:



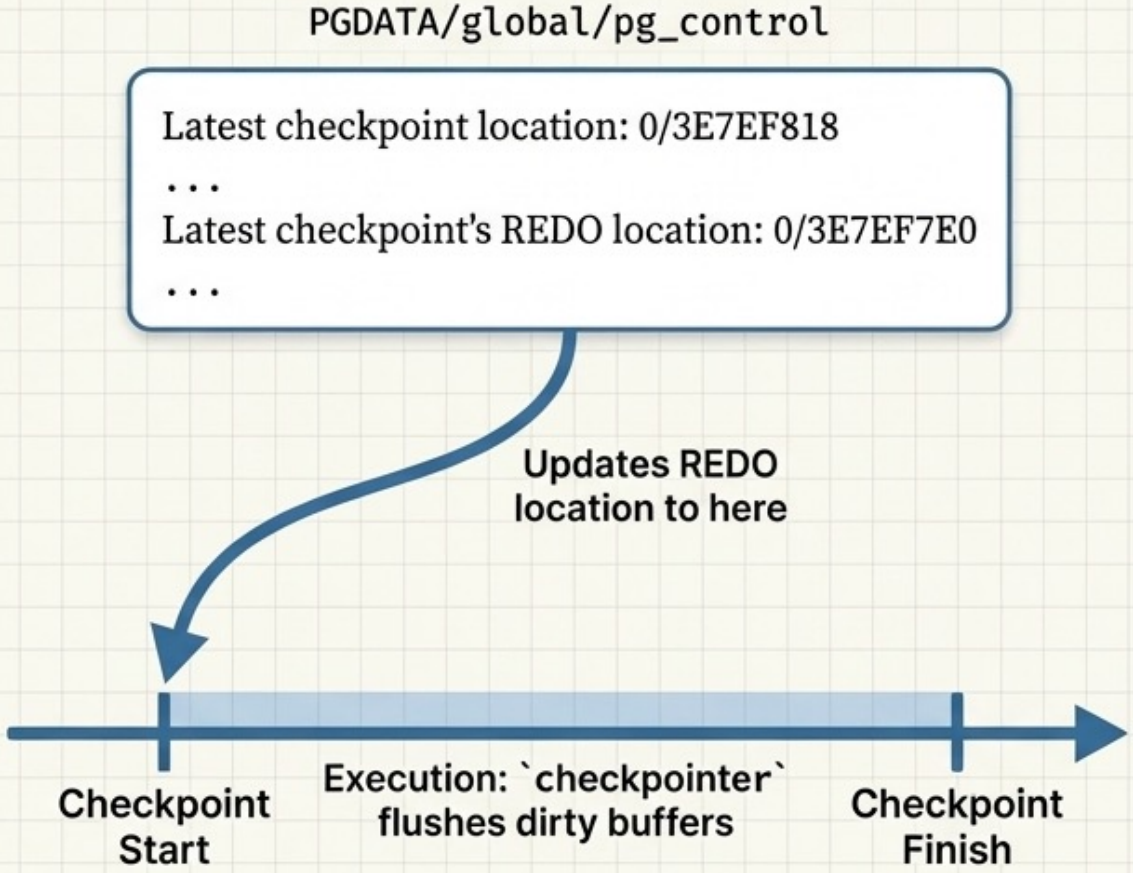
After:



NotebookLM

How a Checkpoint Works: A Coordinated Flush

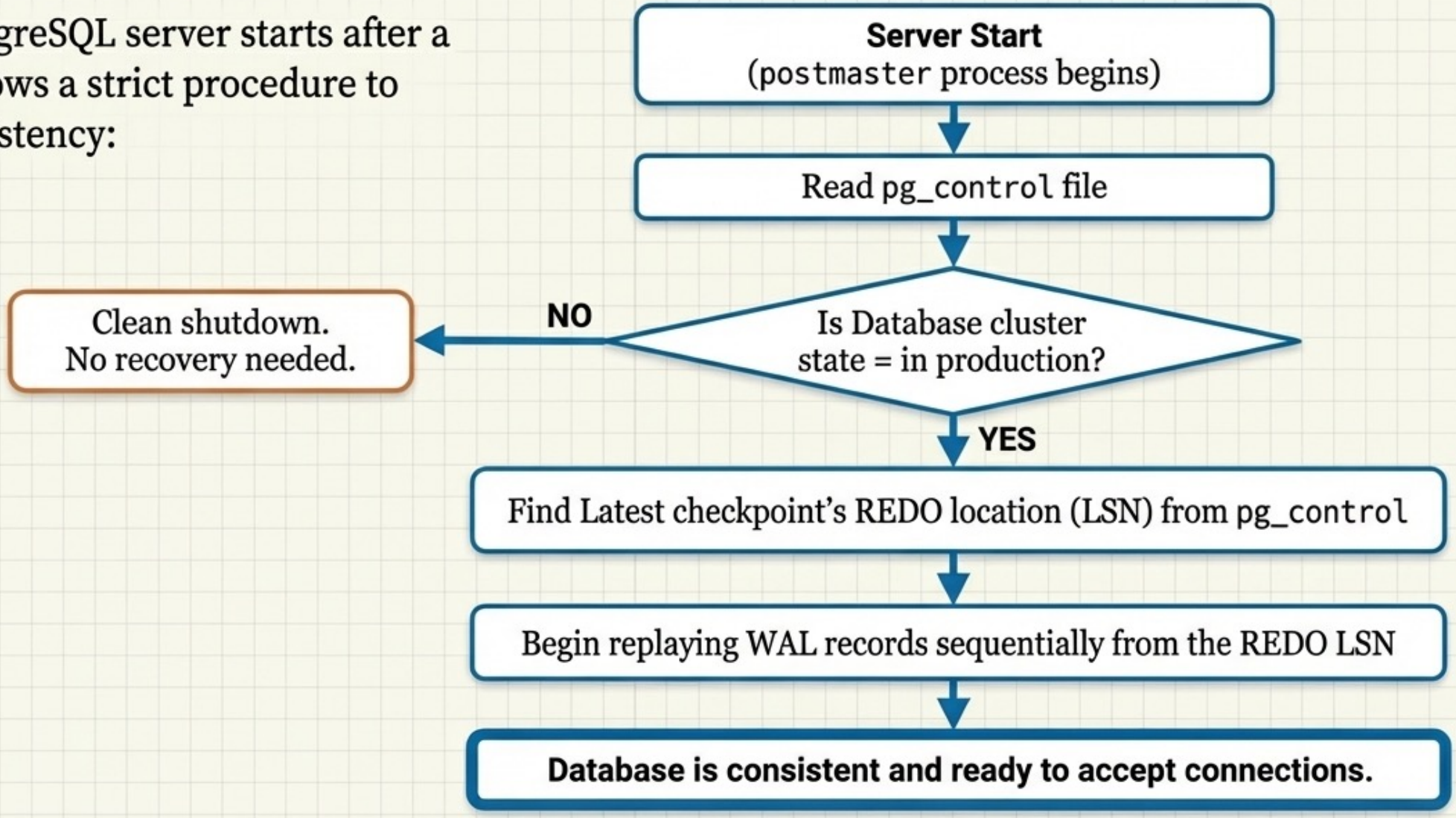
1. **Start:** A "checkpoint start" location (the REDO point) is identified. The **`checkpointer`** process marks all "dirty" buffers in shared memory. This is very fast.
2. **Execution:** The **`checkpointer`** systematically writes all marked dirty buffers to disk. This work is intentionally spread out over time to avoid I/O storms.
3. **Completion:** Once all dirty buffers are on disk, a final **`CHECKPOINT_ONLINE`** record is written to the WAL. The central **`pg_control`** file is then updated with the location of this new, safe REDO point.



NotebookLM

Crash Recovery in Action: Rebuilding a Consistent State

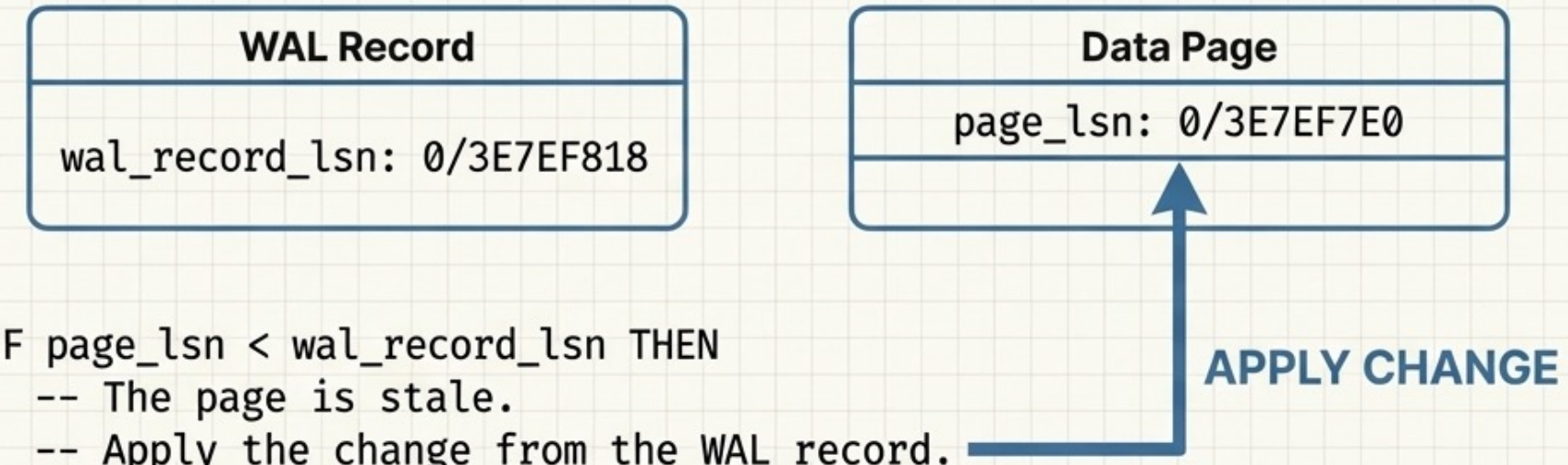
When a PostgreSQL server starts after a crash, it follows a strict procedure to ensure consistency:



NotebookLM

The Replay Logic: Applying the Log Idempotently

For each WAL record read during recovery, PostgreSQL reads the corresponding data page and performs a critical comparison:




```
IF page_lsn < wal_record_lsn THEN
   -- The page is stale.
   -- Apply the change from the WAL record.
ELSE
   -- The page is up-to-date.
   -- Do nothing.
END IF
```

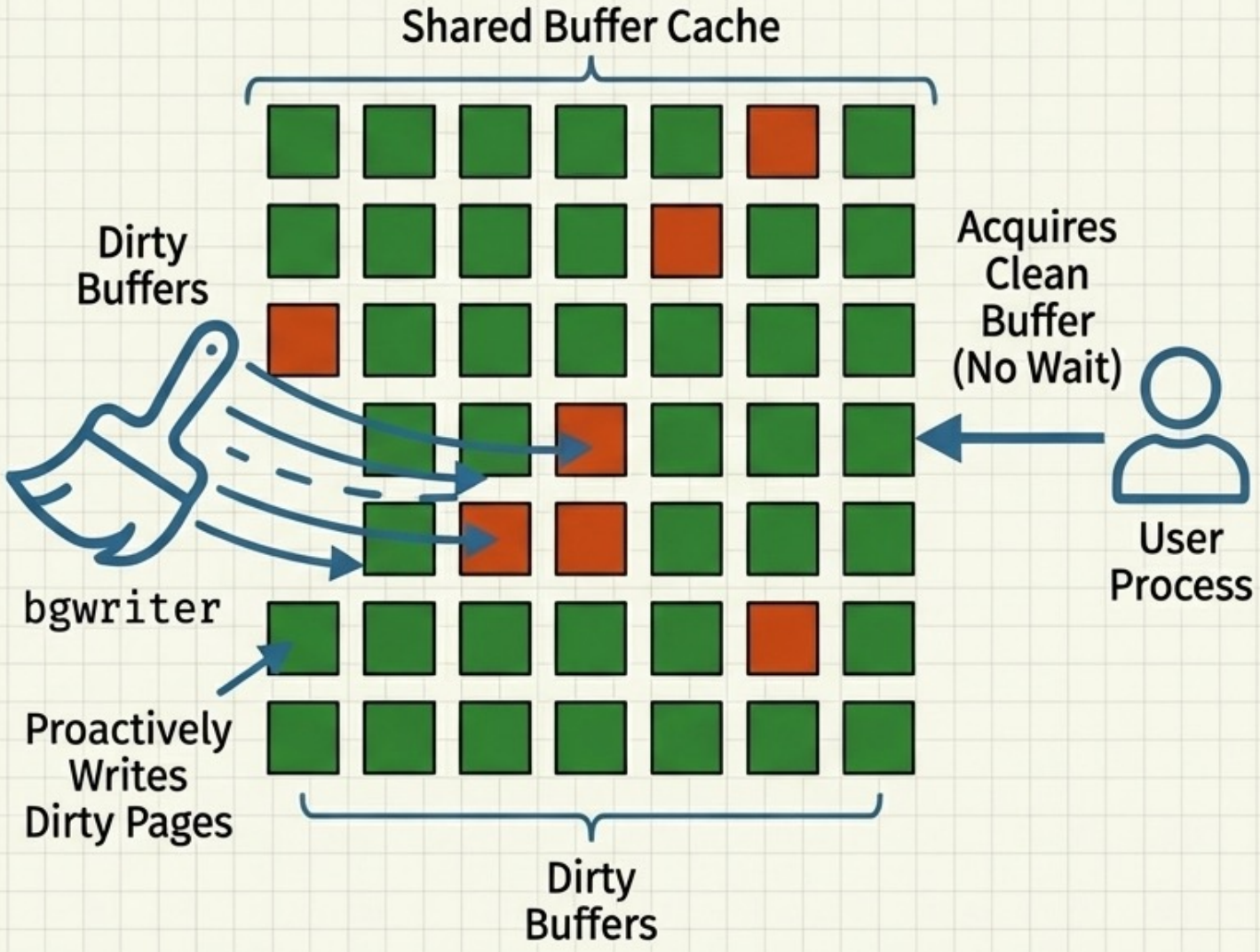
Result: This simple rule ensures that each change is applied exactly once, making the recovery process safe and repeatable (idempotent). This continues until the end of the WAL is reached.

NotebookLM

The Background Writer (bgwriter): Proactively Reducing Latency

- **The Problem it Solves:** If a user process needs a buffer from the cache but finds it is 'dirty', it must wait for that buffer to be written out first. This causes latency.
- **The `bgwriter`'s Role:** A separate background process that complements the `checkpointer`.
- **Job:** Periodically scans the buffer cache for dirty pages that are likely to be needed soon and writes them to disk proactively.

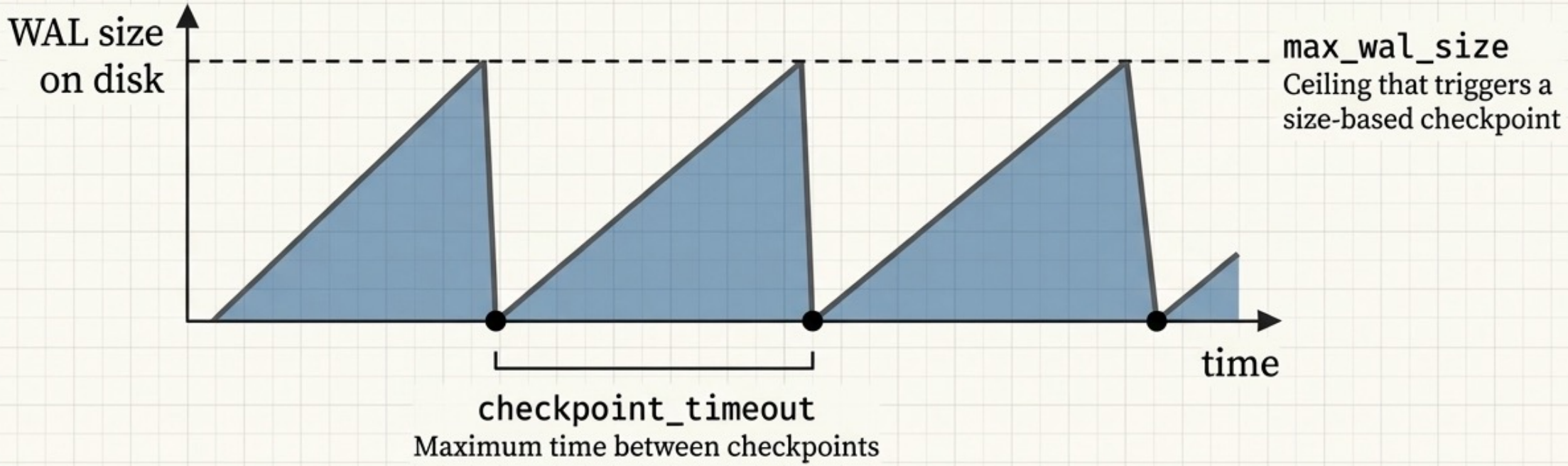
  Goal: To increase the probability that user processes will find 'clean' buffers, minimizing write stalls they would otherwise experience.



NotebookLM

Tuning Checkpoints: Balancing I/O with Recovery Time

The Goal: Smooth out disk I/O to prevent performance-killing bursts of write activity, while controlling the maximum time needed for recovery.



`checkpoint_timeout` (e.g., `` `30min` ``): The primary trigger on a system with a steady workload.

`max_wal_size` (e.g., `` `16GB` ``): A safety valve to handle bursts of activity and keep recovery time bounded.

`checkpoint_completion_target` (e.g., `` `0.9` ``): The fraction of the interval between checkpoints over which to spread the I/O to smooth performance.

NotebookLM

Monitoring Your Strategy with pg_stat_bgwriter

The `pg_stat_bgwriter` view provides essential statistics on checkpoint and background writer activity.

Metric Name	Guideline
`checkpoints_timed`	Triggered by checkpoint_timeout. In a well-tuned system, this should be the dominant number.
`checkpoints_req`	Triggered by max_wal_size. If consistently high, max_wal_size is likely too small, causing frequent, disruptive checkpoints.
`buffers_backend`	Buffers written by user processes. This number should be very low. A high value indicates I/O stalls for users.
`buffers_checkpoint & buffers_clean`	Buffers written by the checkpointer and bgwriter. The vast majority of writes should be handled by these processes.

NotebookLM

WAL: The Engine of Durability and Performance

The Log is Paramount

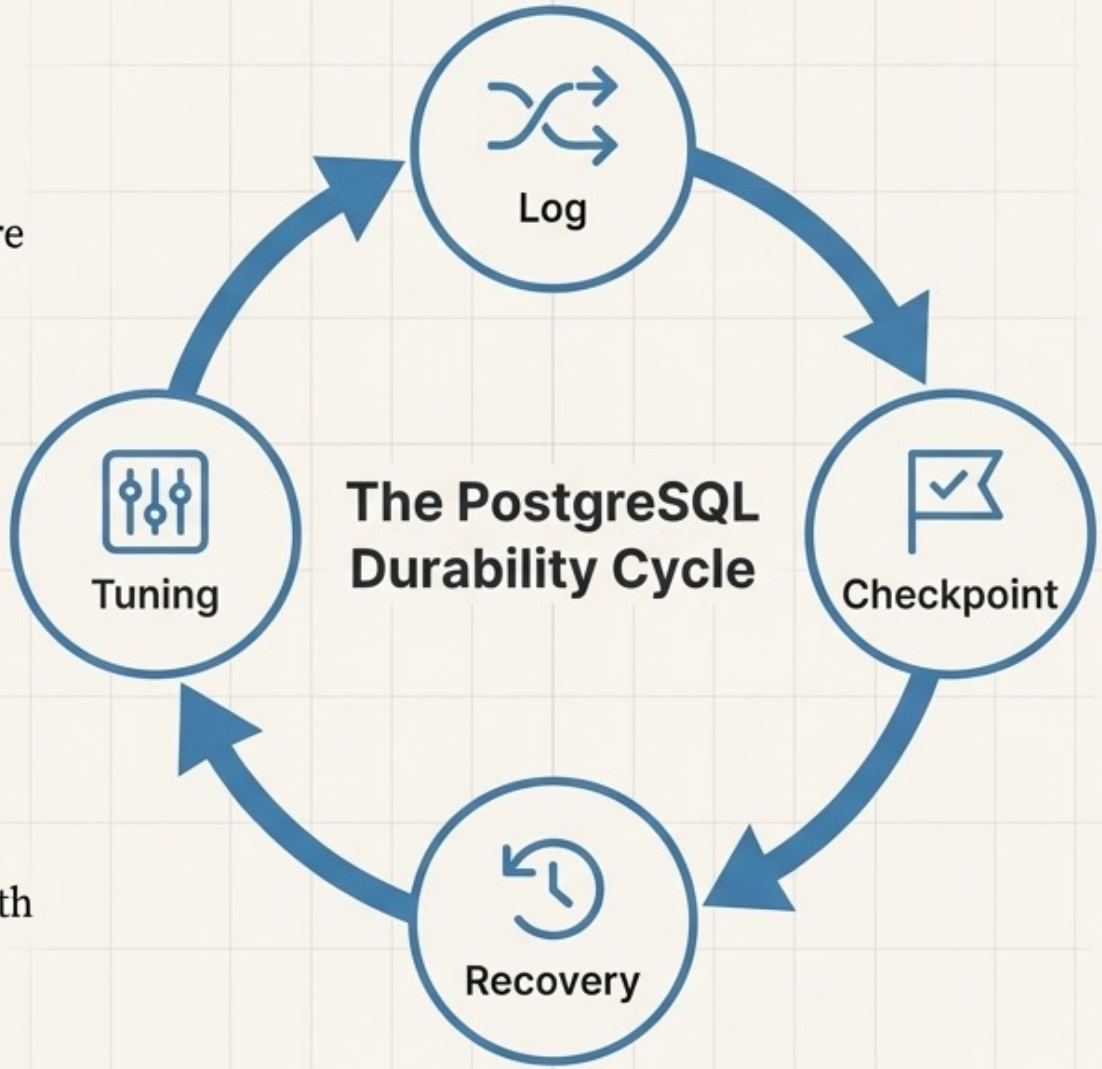
Inter SemiBold

WAL is the single source of truth for durability. Its sequential nature makes it extremely fast.

Checkpoints Manage the Log

Inter SemiBold

Checkpoints create safe recovery points (REDO points), which manage recovery time and allow old WAL segments to be recycled.



Tuning is a Balance

Inter SemiBold

Use `checkpoint_timeout` and `max_wal_size` to balance smooth I/O against your recovery time objectives. Monitor with `pg_stat_bgwriter`.

Recovery is Automatic

Inter SemiBold

PostgreSQL uses the REDO point from the last checkpoint to replay the log and restore a perfectly consistent database state after a crash.

NotebookLM

11. WAL Modes

The Unseen Engine: A Deep Dive into PostgreSQL's Write-Ahead Log Log

Understanding the mechanics of durability, performance, and replication.

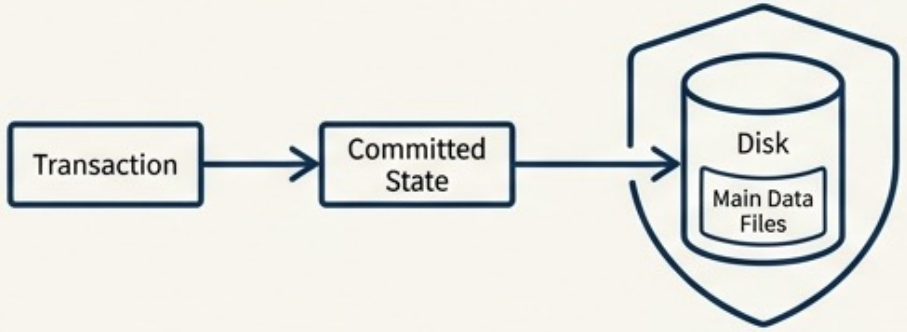
NotebookLM

The Core Dilemma: Durability vs. Performance

At the heart of any ACID-compliant database lies a fundamental challenge:

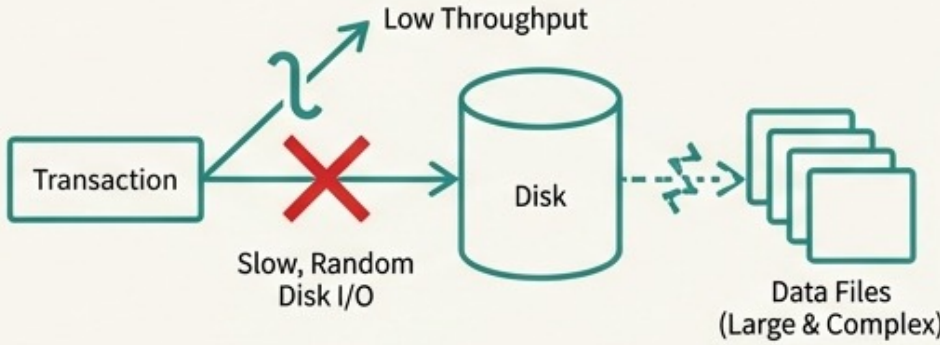
Durability

Durability requires that once a transaction is committed, it must survive any subsequent failure (like a power outage or crash). The simplest way is to write changes directly to the main data files on disk.



Performance

Performance is crippled by this approach. Data files are large and complex, and writing to them involves slow, random disk I/O operations.

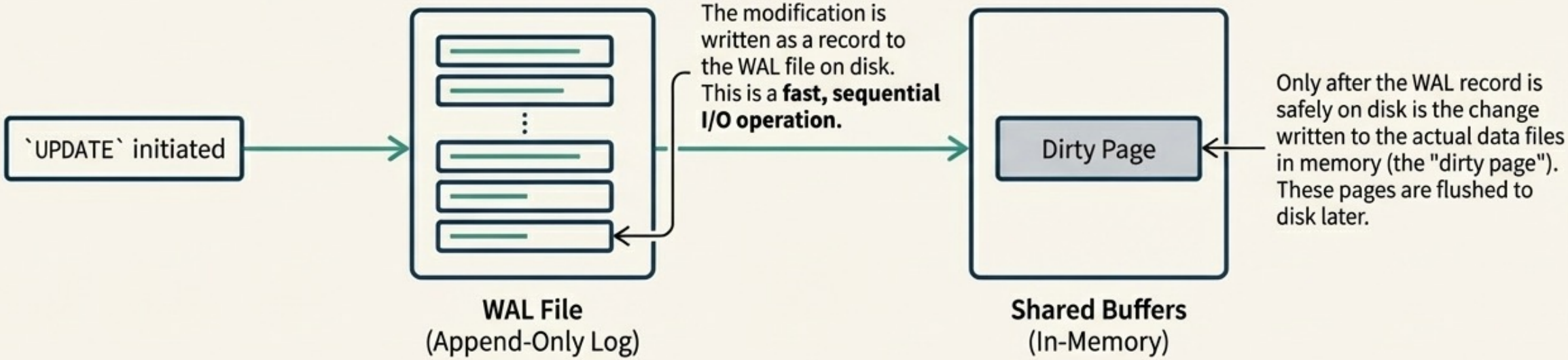


How can PostgreSQL guarantee durability without sacrificing performance?

NotebookLM

The Solution: Write Changes Ahead to a Log

Instead of writing directly to data files, PostgreSQL first records all modifications in a separate, append-only log: the Write-Ahead Log (WAL).



Key Insight: In case of a crash, PostgreSQL replays the WAL from the last checkpoint to restore the database to a consistent state. The slow, random I/O is deferred.

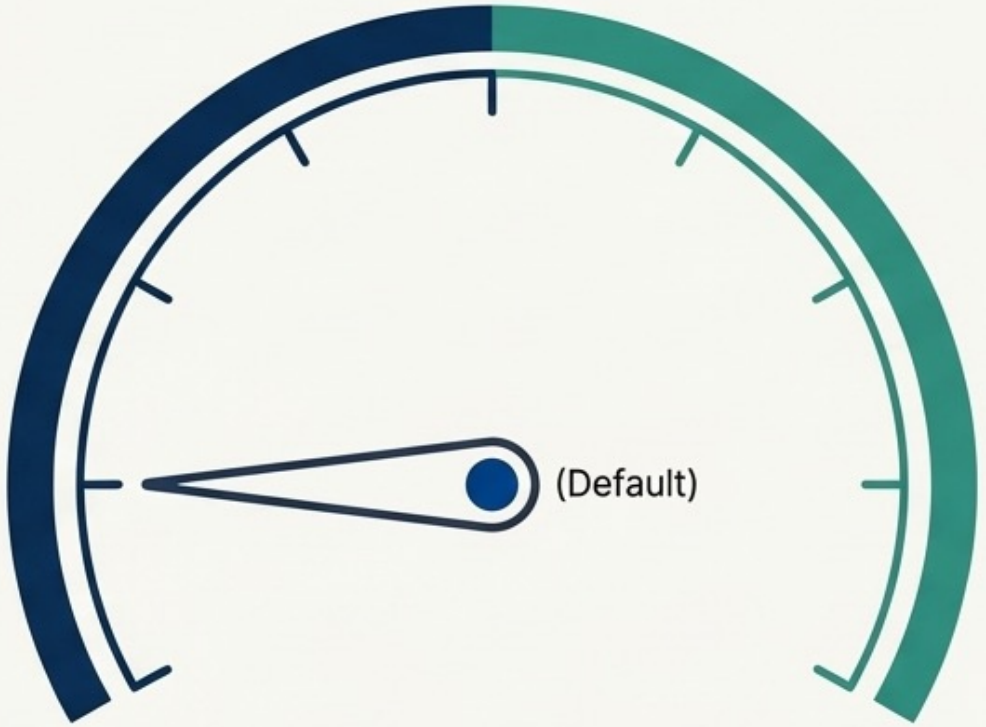
Pro Tip: The sequential nature of WAL I/O makes it a prime candidate for dedicated physical storage. Consider mounting `PGDATA/pg_wal` on its own device.

NotebookLM

The Performance Dial: `synchronous_commit`

This parameter is the most critical tuning knob for WAL behavior. It directly controls the trade-off between transaction latency and durability guarantees. It primarily operates in two modes:

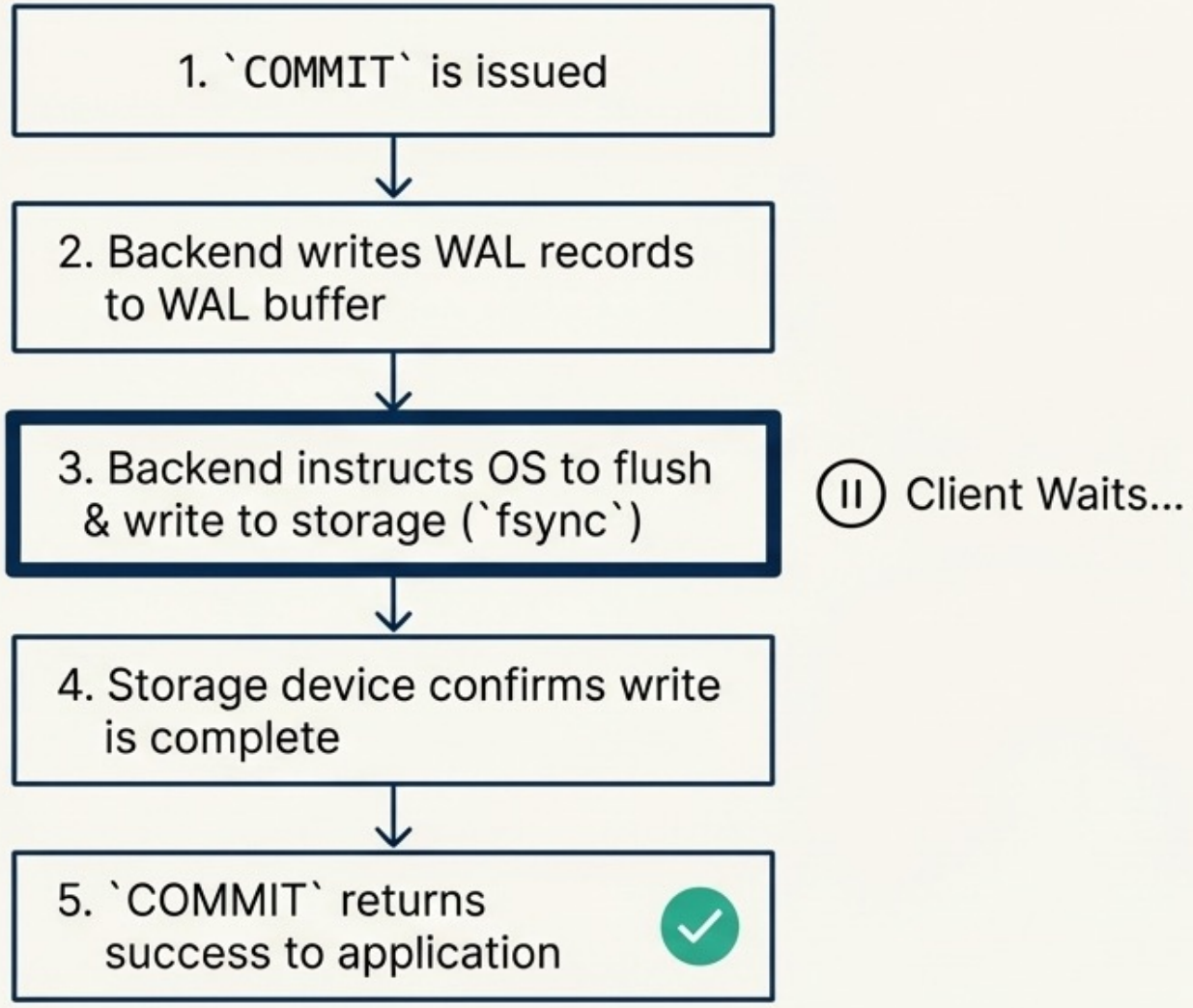
- **`on` (Synchronous):** The transaction waits for its WAL records to be physically written and synced to disk before returning success to the client.
  Maximum safety.
  (Default)



- **`off` (Asynchronous):** The transaction commits as soon as the WAL record is written to the OS cache. It does not wait for the disk sync.
  Maximum performance.

NotebookLM

Synchronous Commit: The Path to Guaranteed Durability



The Bottom Line:

The `fsync' call is the guarantee. It can be slow, as it involves actual I/O, but it ensures that if the client receives a "commit OK," the data is safe even from an immediate OS crash or power loss.

NotebookLM

Optimizing Durability: The Group Commit “Elevator”

Problem

In a high-throughput system, forcing every single transaction to wait for its own `fsync` creates a performance bottleneck.

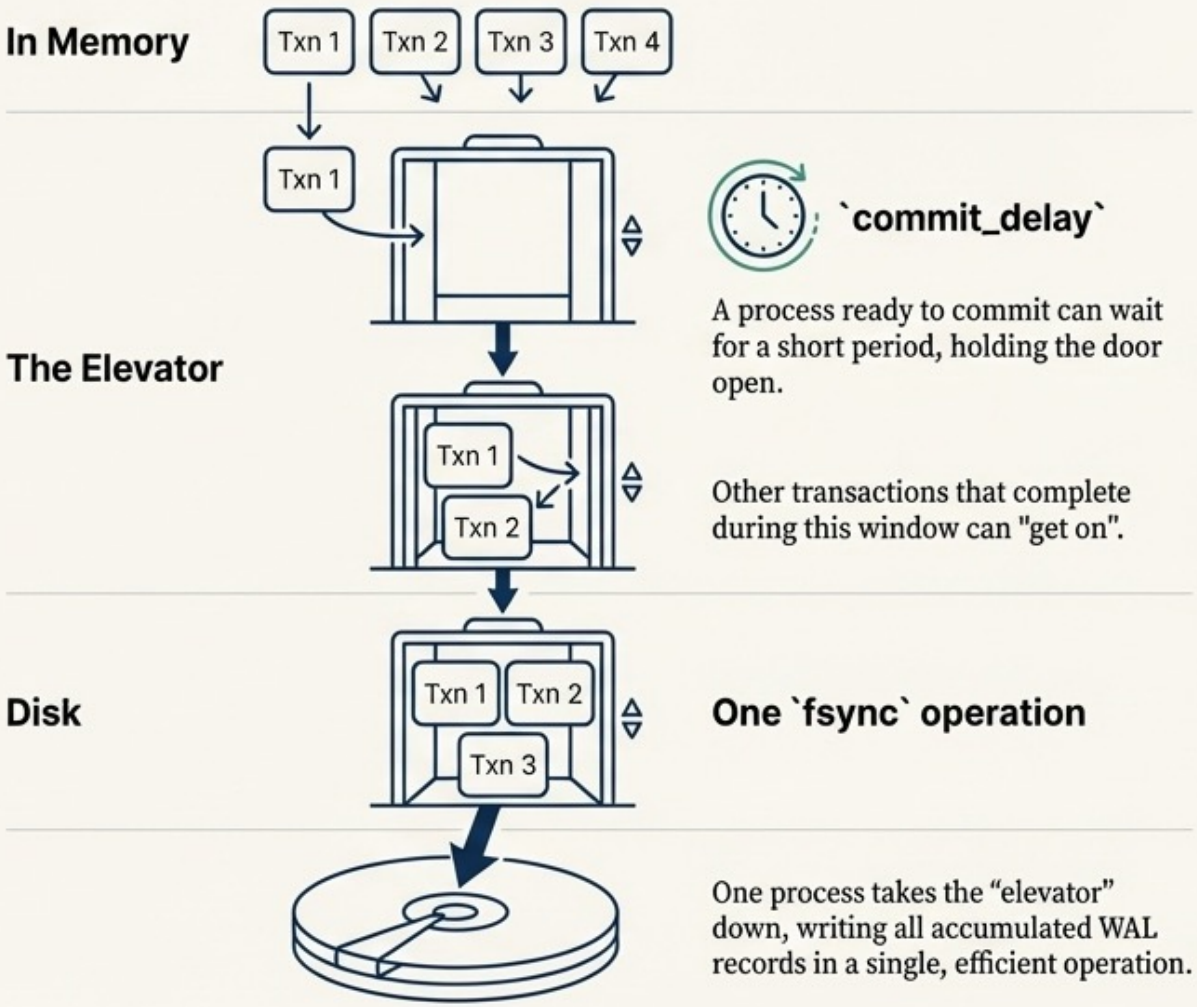
Solution

Group Commit. One process can perform a single `fsync` for multiple transactions that are committing at the same time.

Key Parameters

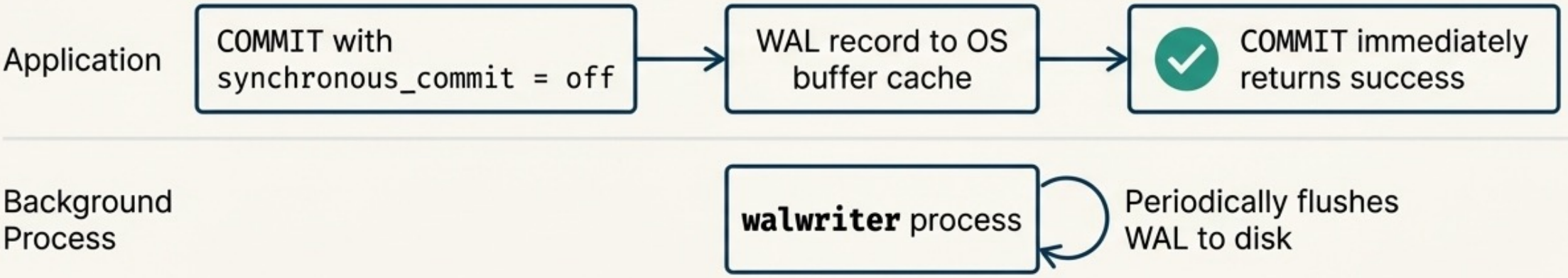
`commit_delay` (default: 0) and `commit_siblings` (default: 5). This is most effective for high-concurrency OLTP workloads.

“Elevator” Analogy for Group Commit



NotebookLM

Asynchronous Commit: Trading Risk for Speed



Tuning the **walwriter**

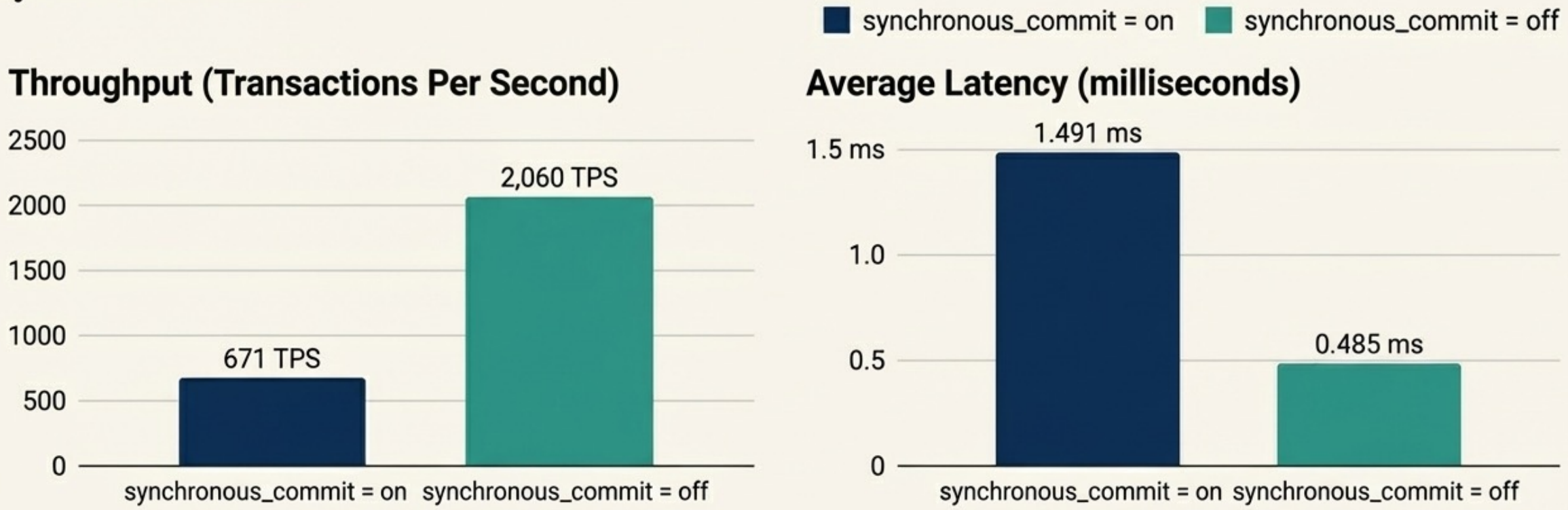
- `wal_writer_delay` (default 200ms): How long the `walwriter` sleeps between flushes.
- `wal_writer_flush_after` (default 1MB): Amount of WAL that triggers a flush.

The Explicit Trade-Off: This mode is significantly faster, but a server crash can result in the loss of transactions that were reported as committed. The maximum window for data loss is roughly **3 × wal_writer_delay** (e.g., 600ms with default settings).

NotebookLM

The Impact in Numbers: A `pgbench` Comparison

A simple TPC-B-like benchmark was run for 30 seconds, first in synchronous mode, then in asynchronous mode.



For short OLTP transactions, switching to asynchronous commit can triple throughput and reduce latency by nearly 70%. This is a powerful tuning option if the application can tolerate the small risk of data loss.

NotebookLM

Beyond Performance: WAL as a Guardian of Data Integrity

WAL's primary job is to ensure crash recovery is always possible, assuming the underlying storage is intact. This requires defending against a hostile environment of hardware and software failures.

We will explore three key threats and their WAL-based defenses:



1. Ambiguous Caching Layers



2. Silent Data Corruption



3. Partial (Non-Atomic) Writes

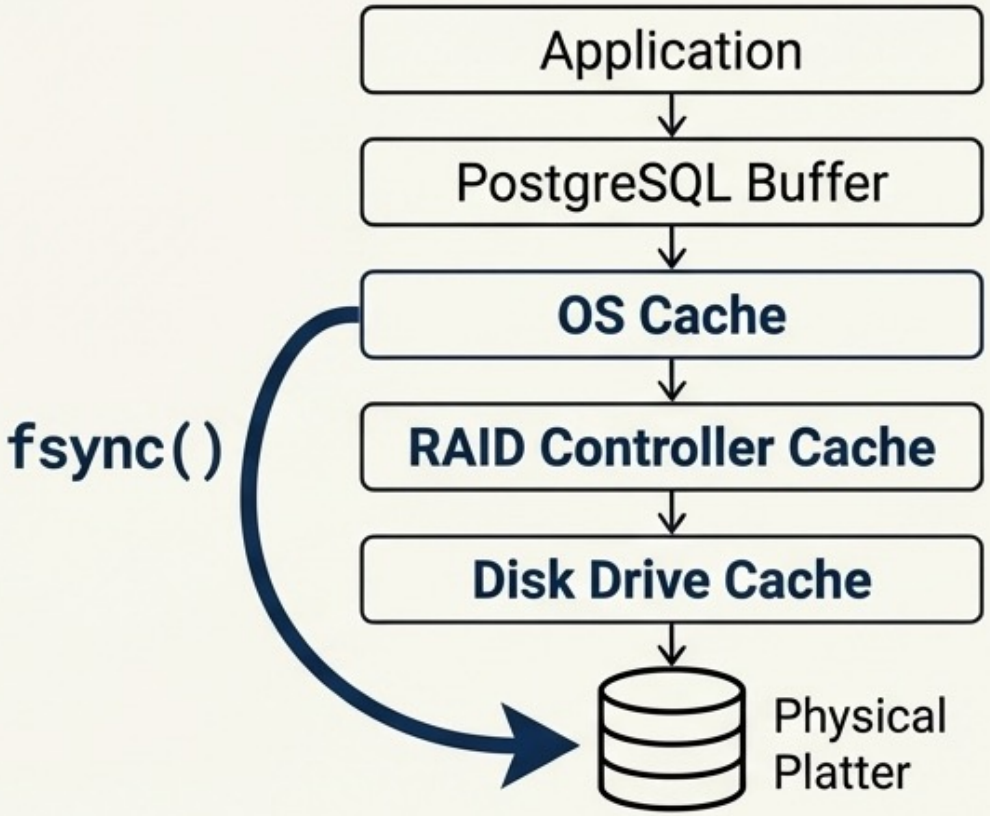
NotebookLM

Threat: Ambiguous Caching.

A ‘write’ command doesn’t mean data is on persistent storage. It can reside in multiple volatile caches.

Defense: `fsync()`.

This system call explicitly commands the OS to flush its buffers all the way down to the persistent storage.



`synchronous_commit = off`

A safe, controlled trade-off. It delays the `fsync` call but still guarantees an eventual consistent state.

`fsync = off`

Disables `fsync` calls entirely. This is **extremely dangerous.** A crash will almost certainly lead to unrecoverable data corruption. Never turn this off in production.

NotebookLM

Threat: Silent Data Corruption. Defense: Checksums.

Threat: Silent Data Corruption.

Hardware is not perfect. Data can be corrupted in memory, in-flight over a cable, or on the disk itself ("bit rot").



Defense: Checksums.

PostgreSQL employs checksums to verify data integrity.

- WAL Records: Always protected by a checksum.
- Data Pages: Can be protected by checksums. This is enabled at cluster initialization (`initdb`) or later with the `pg_checksums` utility. Controlled by the `data_checksums` parameter.



Checksums in Action

If you try to read a corrupted page, you will receive a clear warning, preventing the use of bad data:

```
WARNING: page verification failed, calculated checksum 20397 but
expected 28733
ERROR:  invalid page in block 0 of relation base/16391/16562
```

Recommendation:

Always enable `data_checksums` on production systems. The performance overhead is minor compared to the risk of undetected corruption.

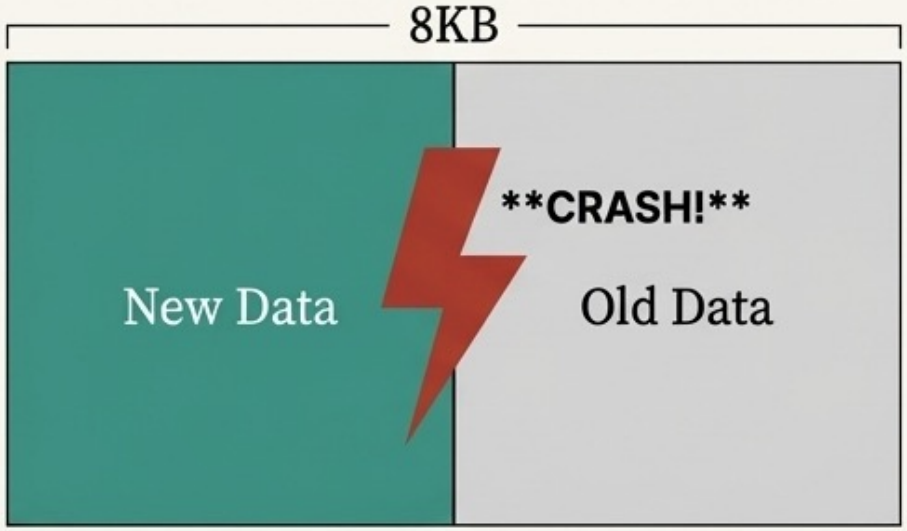
NotebookLM

Threat: Partial Writes (Torn Pages).

A database page is typically 8KB. The OS or hardware may write data in smaller blocks (e.g., 4KB or 512 bytes).

A power failure during a write can leave a page in a corrupt, half-written state.

Applying a normal WAL record to a torn page would only worsen the corruption.



Defense: Full Page Images (FPI).

Controlled by `full_page_writes = on`.

The very first time a data page is modified after a checkpoint, PostgreSQL writes a complete copy (an image) of that page into the WAL stream.

During recovery, if PostgreSQL encounters an FPI, it overwrites the on-disk page with this guaranteed-good copy before applying subsequent WAL records. This repairs any potential “tear.”

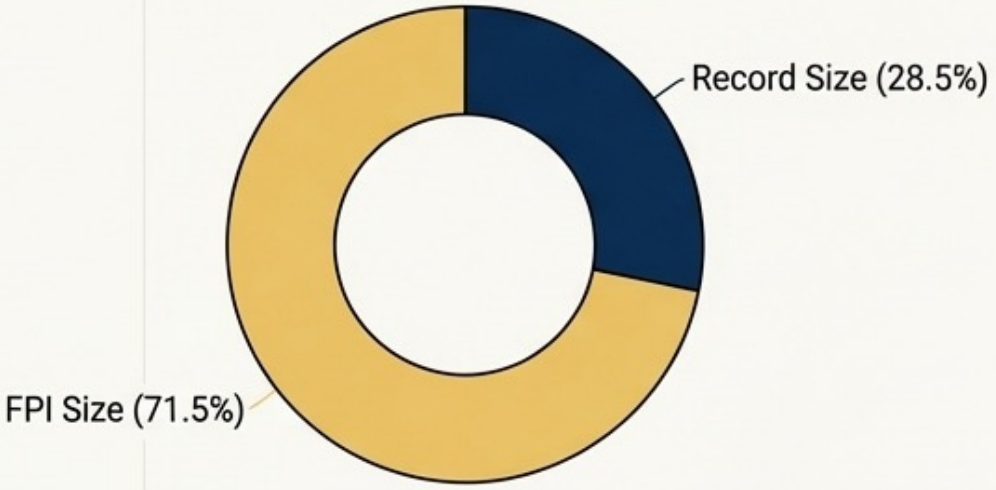
NotebookLM

The Cost of Safety: FPIs and WAL Volume

Full Page Images are essential for safety, but they can dramatically increase the volume of WAL generated, as every 8KB page write becomes an 8KB WAL record.

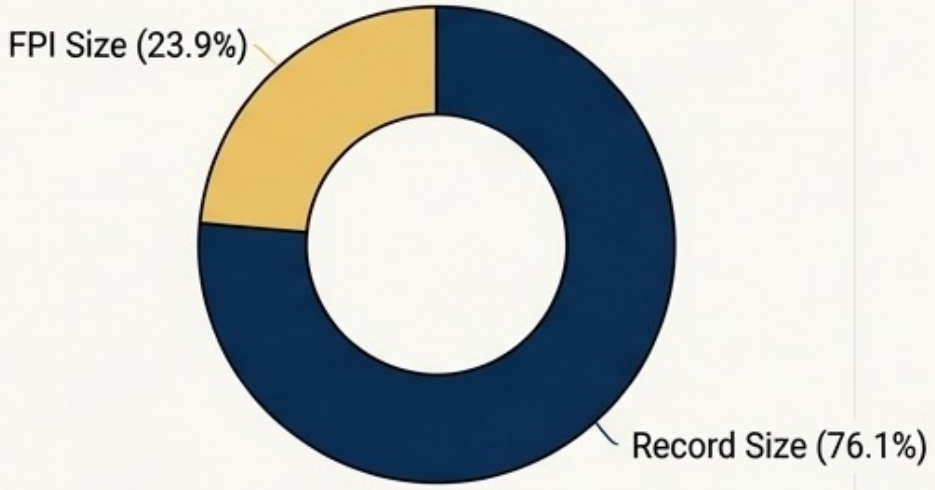
wal_compression = off

Total WAL: 29 MB



wal_compression = on

Total WAL: 10 MB

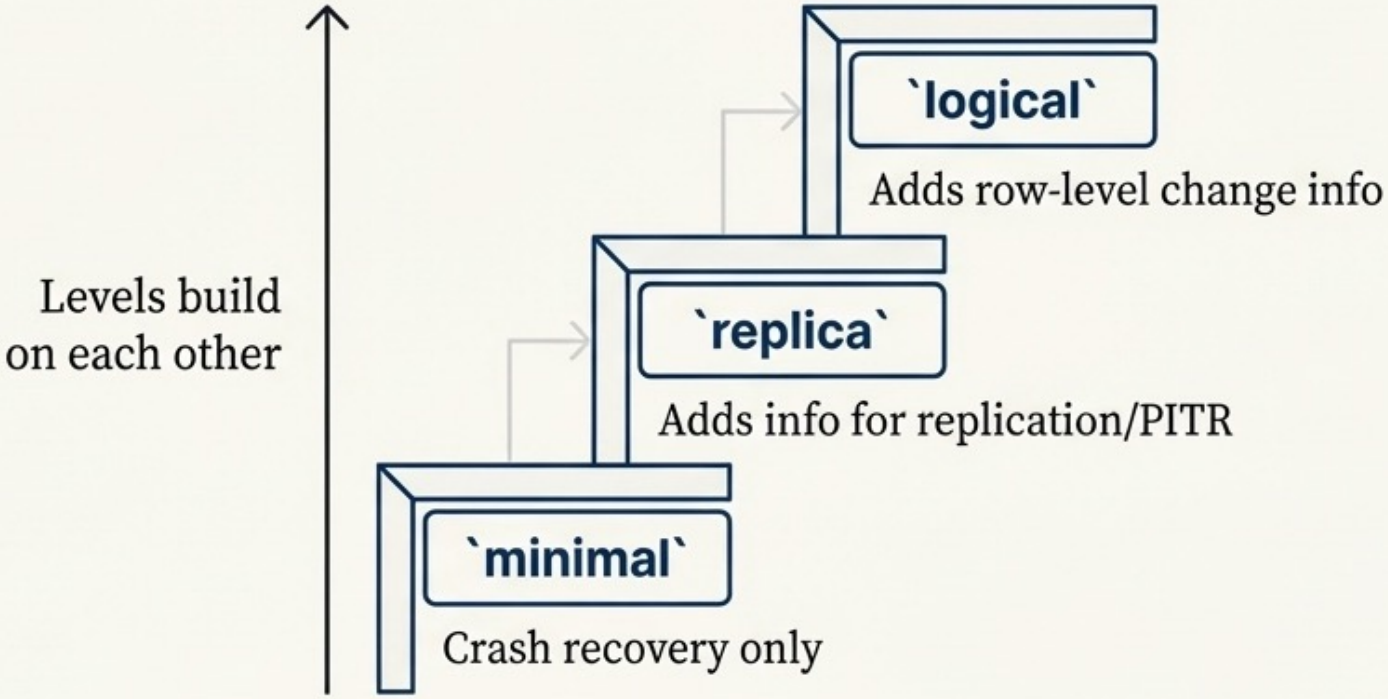


Enabling `wal_compression` reduces total WAL volume by over 65% in this FPI-heavy workload. This is a highly recommended setting if WAL I/O is a bottleneck, trading a small amount of CPU for a large reduction in I/O.

NotebookLM

WAL as a Platform: The Three Levels of Detail

Beyond crash recovery, the information in the WAL stream can be used for replication and data extraction. The `wal_level` parameter controls how much detail is logged.



Important: Changing `wal_level` requires a server restart.

NotebookLM

Level 1: `wal_level = minimal`

Purpose: Provides just enough information for **crash recovery.** Not sufficient for point-in-time recovery (PITR) or streaming replication.

Inside the Log (`pg_waldump` excerpt):

When populating a truncated table with 100,000 rows, the WAL contains:

```
... CREATE base/16391/24784
... UPDATE ... pg_class ...
... INSERT_LEAF ... index ...
... COMMIT ...
```



Notice What's Missing:

There are no records for the 100,000 inserted data rows. The data is written directly to the new file, and only the metadata changes are logged.

Key Optimization

For certain bulk operations (like `CREATE TABLE AS SELECT` or `TRUNCATE` followed by a large `INSERT` in the same transaction), PostgreSQL can skip logging the new data rows entirely if they exceed `wal_skip_threshold` (default 2MB).

NotebookLM

Level 2: `wal_level = replica` (The Default)

Purpose: Logs enough information to support:

- WAL Archiving & Point-in-Time Recovery (PITR)
- Streaming Replication (Physical)

What's Added:

1. All data modifications are logged, with no exceptions for bulk loads.
2. Additional information required by standbys is periodically logged, such as running transaction snapshots (`RUNNING_XACTS`) and exclusive lock information (`LOCK`).

Inside the Log (`pg_waldump` excerpt):
Repeating the `TRUNCATE` and `INSERT` operation now shows:

```
... Standby: LOCK ...
... Heap: INSERT+INIT ...
... Standby: RUNNING_XACTS ...
... COMMIT ...
```

The presence of `Standby` records and the `INSERT+INIT` record for the data itself are the key differences.

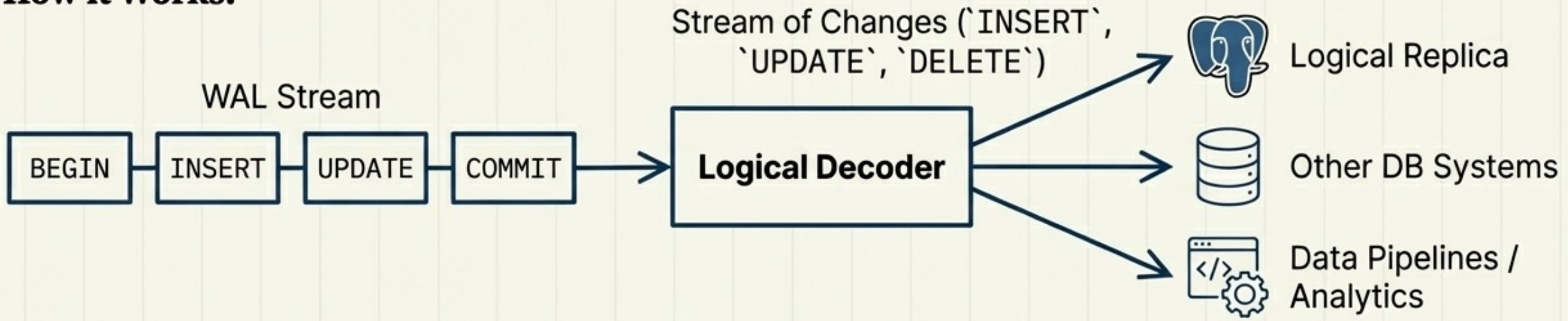
NotebookLM

Level 3: `wal_level = logical`

Purpose:

- **Logical Decoding:** The process of extracting a coherent, easy-to-interpret stream of changes from the WAL.
- **Logical Replication:** Replicating data changes to other PostgreSQL instances (or even different database systems) based on their logical content (e.g., a stream of `INSERT`, `UPDATE`, `DELETE` statements), not just their physical block changes.

How it Works:

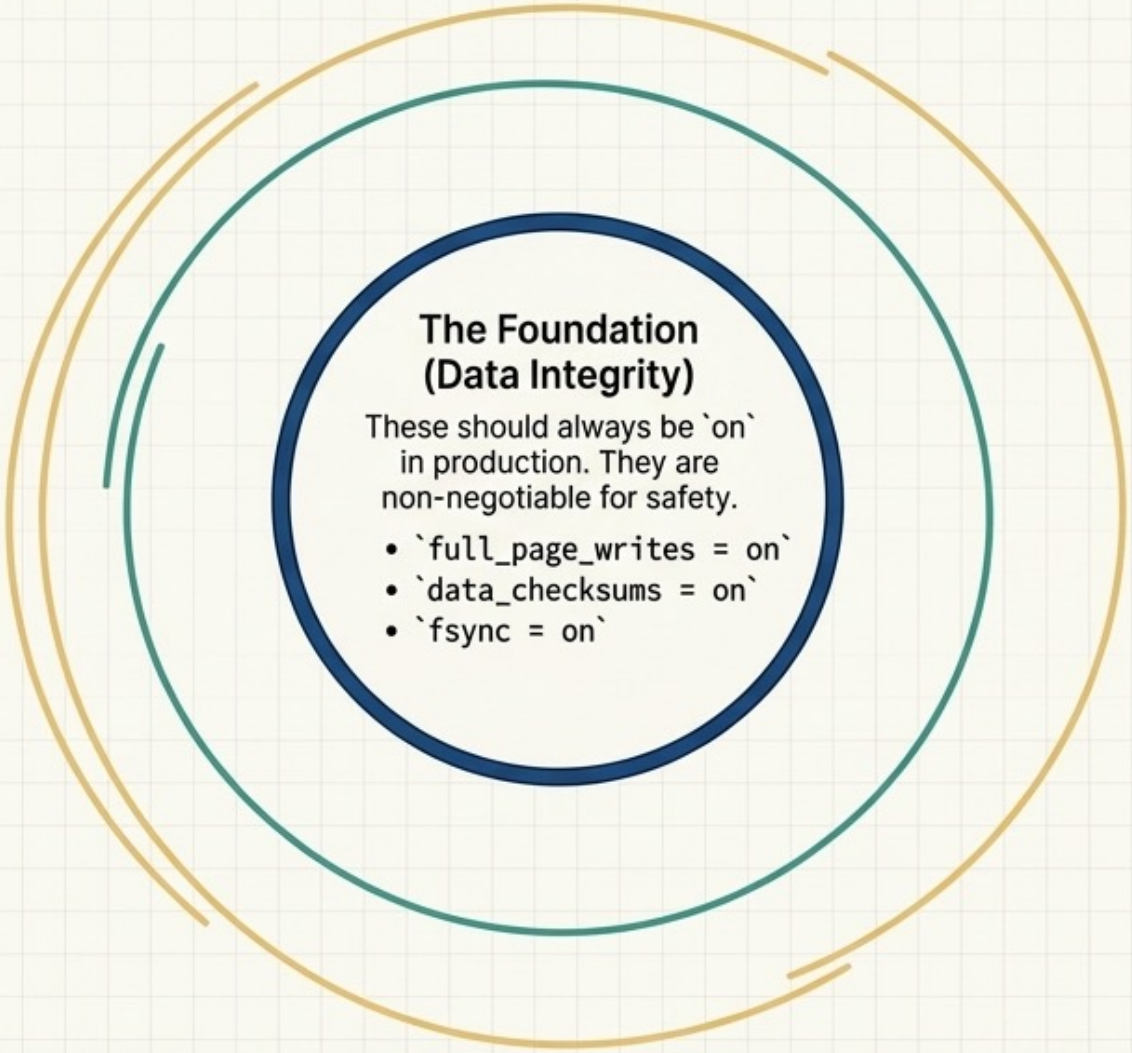


Use Case: This is the required setting for Change Data Capture (CDC) pipelines.

NotebookLM

A Unified Model for WAL Configuration

Choosing the right settings involves balancing capability, durability, and performance. Think of it in layers:



The Goal (Capability & Durability)

- **`wal_level`**: What do you need WAL to *do*?
  - **`replica`**: The robust default for any system needing backups and/or replication.
  - **`logical`**: For Change Data Capture (CDC) and logical replication.
  - **`minimal`**: Only for special cases like ephemeral bulk data loading systems where backups are not needed.

The Dials (Performance Tuning)

- **`synchronous_commit`**: The main performance dial.
  - **`on`**: The safe default.
  - **`off`**: For workloads that can tolerate a minimal data loss window for a large performance gain. Can be set per-transaction.
- **`wal_compression`** & **`commit_delay`**: Fine-tuning optimizations for I/O-bound or high-concurrency OLTP systems.

NotebookLM

12. Relation-Level Locks



The Silent Conductor

Mastering PostgreSQL's Locking Symphony

An in-depth guide to understanding and mastering table-level locking in PostgreSQL.

NotebookLM

The Problem: A Collision of Processes

Locks exist to control concurrent access to shared resources.

Concurrency

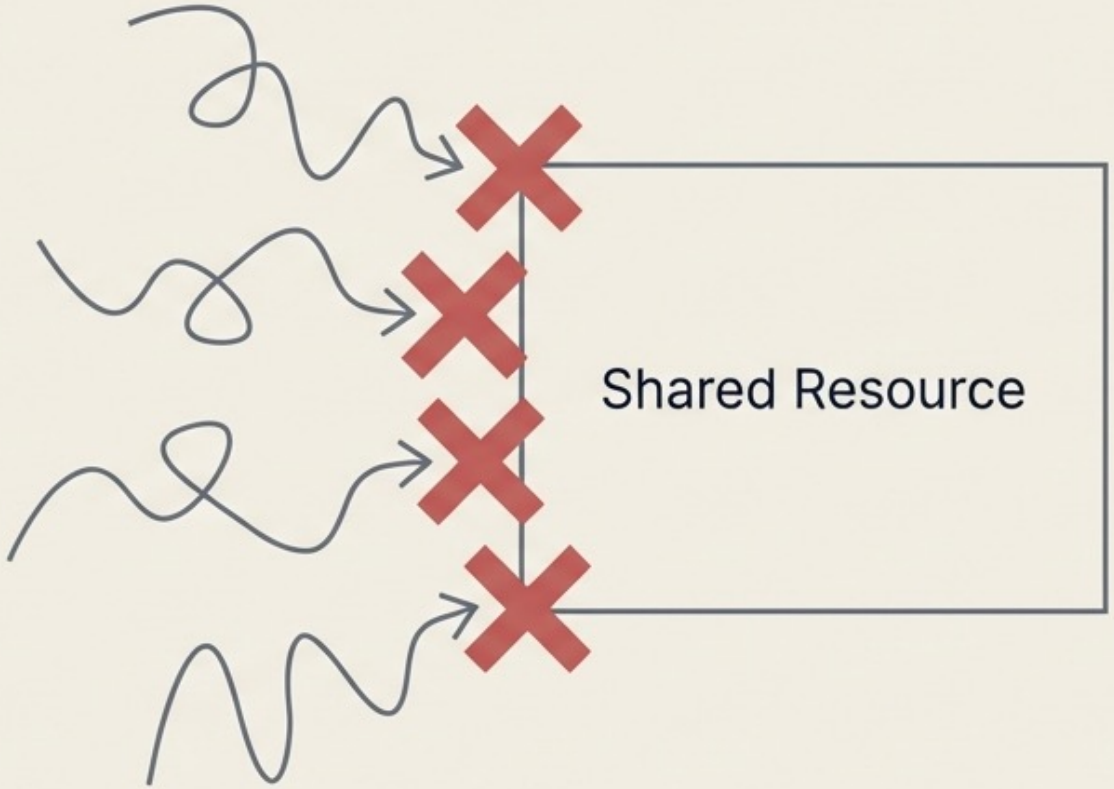
Multiple processes attempting to access the same resource (e.g., a table, a row) at the same time.

The Conflict

Without a control system, simultaneous read and write operations would lead to data corruption and inconsistency.

The Solution

Before accessing a resource, a process must acquire a lock. When finished, it releases the lock, allowing others to proceed in an orderly fashion.



NotebookLM

The Conductor's Toolkit: Two Key Properties of Locks

Granularity (The 'Scope' of the Lock)

- **Coarse-Grained** (e.g., **Table Lock**): Locks the entire object. Simpler to manage but can block unrelated operations.



- **Fine-Grained** (e.g., Row **Lock**): Locks a smaller component. Allows for much higher concurrency but increases management overhead.



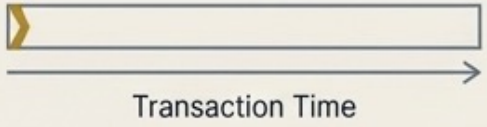
PostgreSQL can use lock escalation to replace numerous fine-grained locks with a single coarse-grained one to save memory.

Duration (The 'Lifetime' of the Lock)

- **Long-Term**: Held for a significant duration, often until the end of a transaction. Protects resources like relations and rows. These are the 'heavyweight' locks we will focus on.



- **Short-Term**: Held for microseconds. Protects internal memory structures. Managed automatically and not visible to users.



NotebookLM

The Toolkit: Lock Modes Define Compatibility

A lock isn't just "on" or "off." Its *mode* determines what other operations can happen concurrently.

The Fundamental Modes

- **Shared Mode:** Allows multiple processes to hold the lock simultaneously. Used for reading.
- **Exclusive Mode:** Only one process can hold the lock. Blocks all other processes. Used for writing.

The Rulebook

The compatibility of different modes is defined by a compatibility matrix.

	Shared	Exclusive
Shared	✓	✗
Exclusive	✗	✗

NotebookLM

The Ensemble: PostgreSQL's Heavyweight Locks

Heavyweight locks are the long-term, object-level locks that protect relations (tables, indexes), rows, and other database objects.

Key Characteristics

- **Location**: Reside in the server's shared memory.
- **Visibility**: Can be inspected via the `pg_locks` view.
- **Automation**: Mostly acquired and released automatically by PostgreSQL during a transaction.
- **Infrastructure**: Feature a full support system including wait queues and automatic deadlock detection.



System Limits

The total number of heavyweight locks is limited by:

`max_locks_per_transaction` (default: 64)

×

`max_connections` (default: 100)

Changing these parameters requires a server restart.

NotebookLM

The Sheet Music: 8 Distinct Table-Level Lock Modes

PostgreSQL offers eight different lock modes for relations to maximize the number of concurrent operations possible.

1. `ACCESS SHARE`
2. `ROW SHARE`
3. `ROW EXCLUSIVE`
4. `SHARE UPDATE EXCLUSIVE`
5. `SHARE`
6. `SHARE ROW EXCLUSIVE`
7. `EXCLUSIVE`
8. `ACCESS EXCLUSIVE`

You don't need to memorize every mode, but understanding their general purpose and conflicts is crucial for diagnosing performance issues. The compatibility matrix is your reference.

NotebookLM

Reading the Score: The Table Lock Compatibility Matrix

This matrix shows which lock modes can coexist. An 'X' indicates a conflict. The rightmost column shows common SQL commands that acquire each lock mode.

	AS	RS	RE	SUE	S	SRE	E	AE	Acquired By Commands Like...
Access Share								X	SELECT
Row Share							X	X	SELECT FOR UPDATE/SHARE
Row Exclusive				X	X	X	X	X	INSERT, UPDATE, DELETE
Share Update Exclusive			X	X	X	X	X	X	VACUUM, CREATE INDEX CONCURRENTLY
Share			X	X		X	X	X	CREATE INDEX
Share Row Exclusive			X	X	X	X	X	X	CREATE TRIGGER
Exclusive		X	X	X	X	X	X	X	REFRESH MAT. VIEW CONCURRENTLY
Access Exclusive	X	X	X	X	X	X	X	X	DROP, TRUNCATE, VACUUM FULL, LOCK TABLE, REFRESH MAT. VIEW

NotebookLM

From Score to Performance: Locks in Common Scenarios



The Reader (SELECT)

- Command: `SELECT ...`
- Lock Mode: `Access Share`
- Impact: The weakest lock. Compatible with all modes except `Access Exclusive`. This is why reading rarely blocks other operations (and is rarely blocked), except by commands like `DROP TABLE`.



The Writer (INSERT, UPDATE, DELETE)

- Command: `UPDATE accounts SET ...`
- Lock Mode: `Row Exclusive`
- Impact: The first mode that conflicts with itself. Prevents other write operations from running on the same table concurrently, but still allows `SELECT`s.



The Maintainer (CREATE INDEX)

- Command: `CREATE INDEX ...`
- Lock Mode: `Share`
- Impact: Blocks all write commands (`INSERT`, `UPDATE`, `DELETE`), but allows reads (`SELECT`). This is why index creation can cause application pauses.
- Alternative: `CREATE INDEX CONCURRENTLY` uses the weaker `Share Update Exclusive` lock, which allows writes but takes longer.

NotebookLM

The Performance: The Wait Queue in Action

The Concept

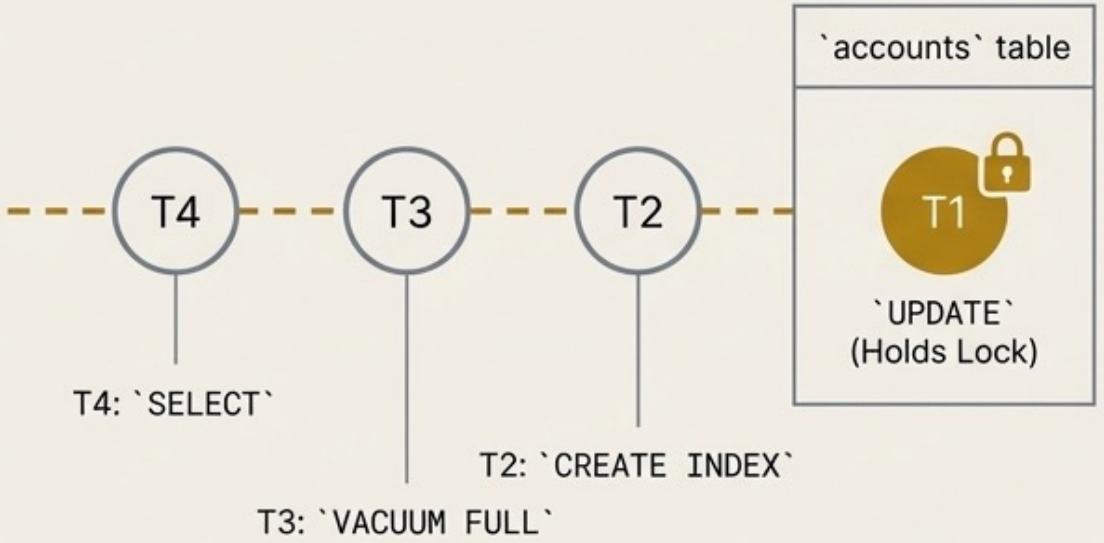
When a process requests a lock that conflicts with a held lock *or* a lock requested by a process already in the queue, it enters a "wait queue."

The Rule

The queue is *fair* (First-In, First-Out). Processes are served in the order they arrive. This prevents process starvation.

Our Scenario

We will observe four transactions competing for a lock on the `accounts` table.



NotebookLM

A Transaction Acquires a Lock

Transaction 1 (PID: 28980) runs an `UPDATE` command.

```
-- Session 1
BEGIN;
UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;
```

The Result: T1 acquires `RowExclusiveLock` on the `accounts` table. This lock is granted immediately.

```
-- Querying pg_locks for PID 28980
 locktype | lockid   | mode             | granted
----------+----------+------------------+---------
 relation | accounts | RowExclusiveLock | t
...
```



NotebookLM

A Queue Begins to Form

Transaction 2 (PID: 29459)

attempts to create an index.

- **Command:** `CREATE INDEX ON accounts(client);`
- **Requested Lock:** `ShareLock`. This conflicts with T1's `RowExclusiveLock`.
- **Result:** T2 enters the wait queue.

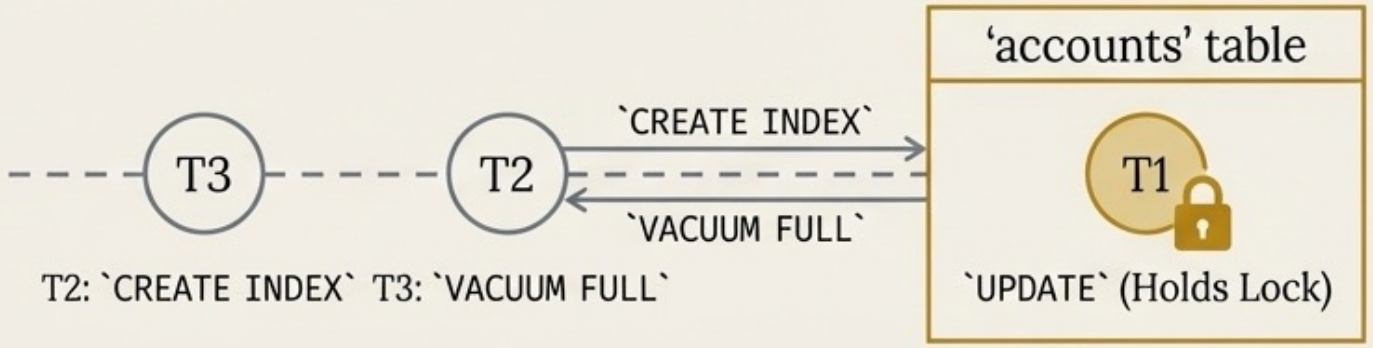
```
-- For PID 29459
 locktype |  lockid  |    mode    | granted
----------+----------+------------+---------
 relation | accounts | ShareLock  | f
```

Transaction 3 (PID: 29662)

attempts a full vacuum.

- **Command:** `VACUUM FULL accounts;`
- **Requested Lock:** `AccessExclusiveLock`. This conflicts with T1's lock *and* T2's requested lock.
- **Result:** T3 enters the wait queue behind T2.

```
-- For PID 29662
 locktype |  lockid  |         mode         | granted
----------+----------+----------------------+---------
 relation | accounts | AccessExclusiveLock  | f
```



NotebookLM

The Fair Queue is Uncompromising

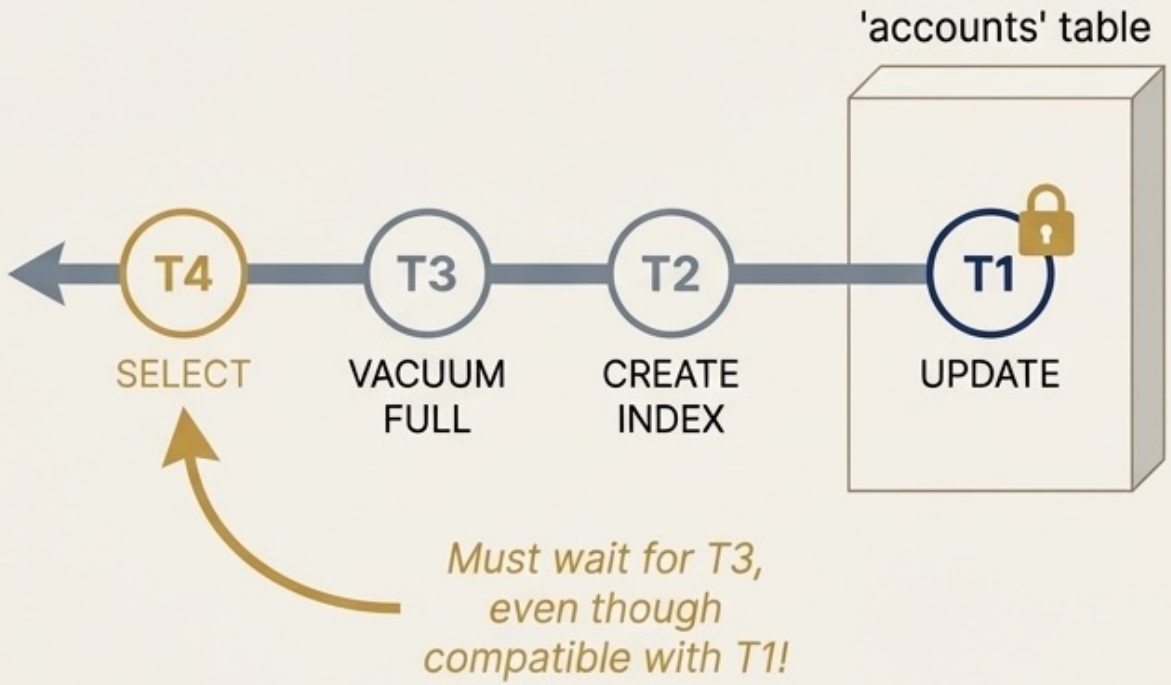
Transaction 4 (PID: 29872) runs a simple `SELECT` query.

- **Command:** `SELECT * FROM accounts;`
- **Requested Lock:** `AccessShareLock`.

The Crucial Insight
The requested `AccessShareLock` *is* compatible with T1's held `RowExclusiveLock`. **However**, T4 must wait because T3 is ahead of it in the queue, and T3's requested `AccessExclusiveLock` conflicts with T4's `AccessShareLock`.

The rule is absolute: You wait your turn, regardless of compatibility with the current lock holder.

```
-- For PID 29872
 locktype |  lockid  |      mode       | granted
----------+----------+-----------------+---------
 relation | accounts | AccessShareLock | f
```



NotebookLM

Diagnosing the Symphony: Viewing the Lock Chain

The `pg_blocking_pids()` function shows which process IDs are blocking a given process.

Querying the Wait Chain

```
SELECT pid, pg_blocking_pids(pid) FROM pg_stat_activity
WHERE state = 'active' AND wait_event_type = 'Lock';
```

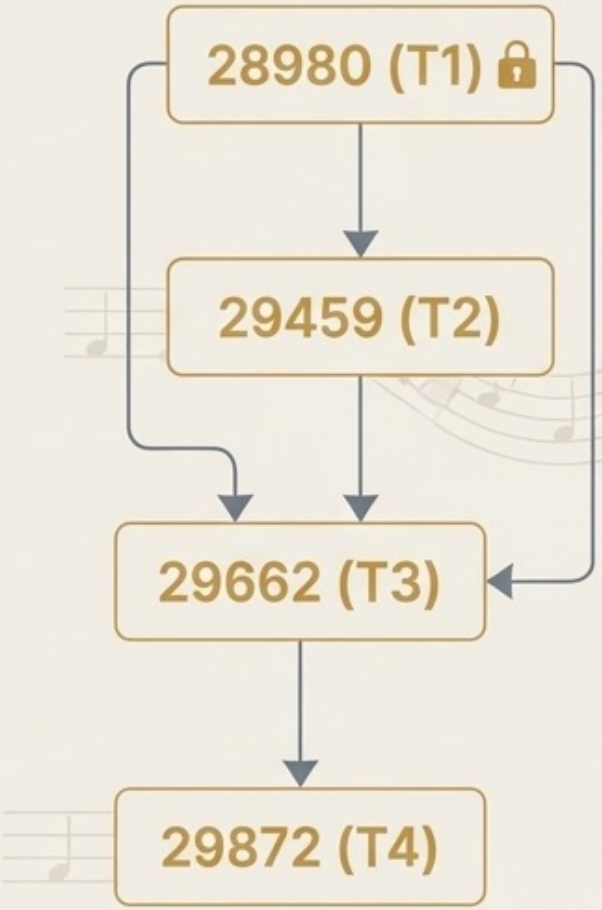
The Resulting Picture

PID 29459 (CREATE INDEX) is blocked by **{28980}**.

PID 29662 (VACUUM FULL) is blocked by **{28980, 29459}**.

PID 29872 (SELECT) is blocked by **{29662}**.

This output clearly maps the chain of dependencies that the **fair queue has** created.



NotebookLM

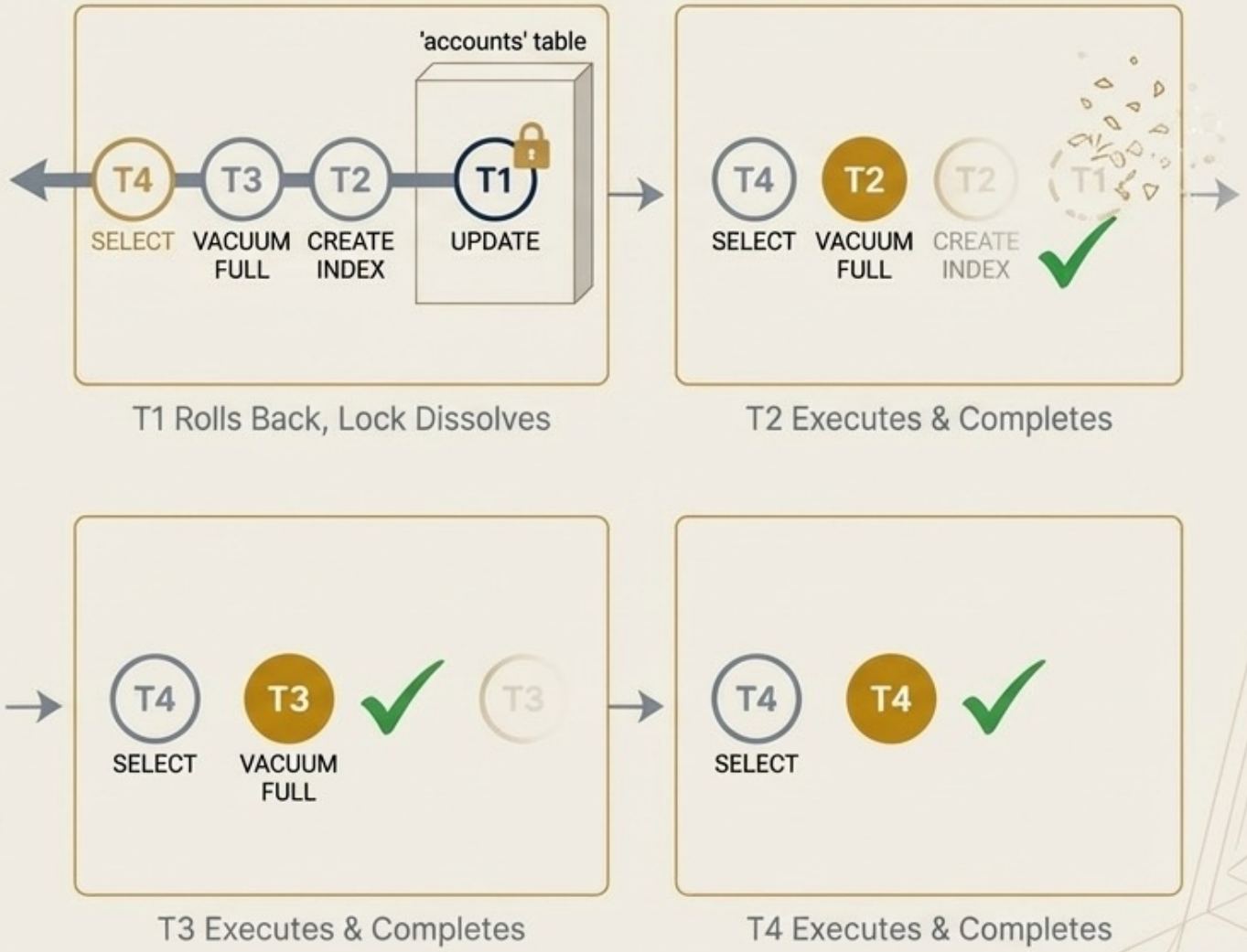
The Conductor's Final Bow: Unlocking the Queue

The Trigger
Transaction 1 (PID **28980**) issues a `ROLLBACK` (or `COMMIT`).

The Cascade

1. T1 releases all its locks.
2. T2 (`CREATE INDEX`), now at the front of the queue, acquires its `ShareLock` and runs to completion.
3. Upon T2's completion, T3 (`VACUUM FULL`) acquires its `AccessExclusiveLock` and runs.
4. Finally, T4 (SELECT) acquires its `AccessShareLock` and executes.

The Result
All transactions are executed sequentially in the order they were queued, ensuring both data integrity and fairness.



NotebookLM

Mastering the Symphony: Core Principles of PostgreSQL Locking



1. The Conductor's Purpose

Locks are not just blockers; they are an orchestration system designed to ensure data integrity while maximizing concurrency.



2. A Rich Toolkit

PostgreSQL's eight table-level lock modes provide a nuanced toolset. Understanding which commands acquire which locks is key to predicting application behavior.



3. The Inviolable Rule

The wait queue is strictly fair (FIFO). A request waits not only for the current lock holder but for everyone ahead of it in the queue. This prevents starvation and is a critical concept for performance tuning.

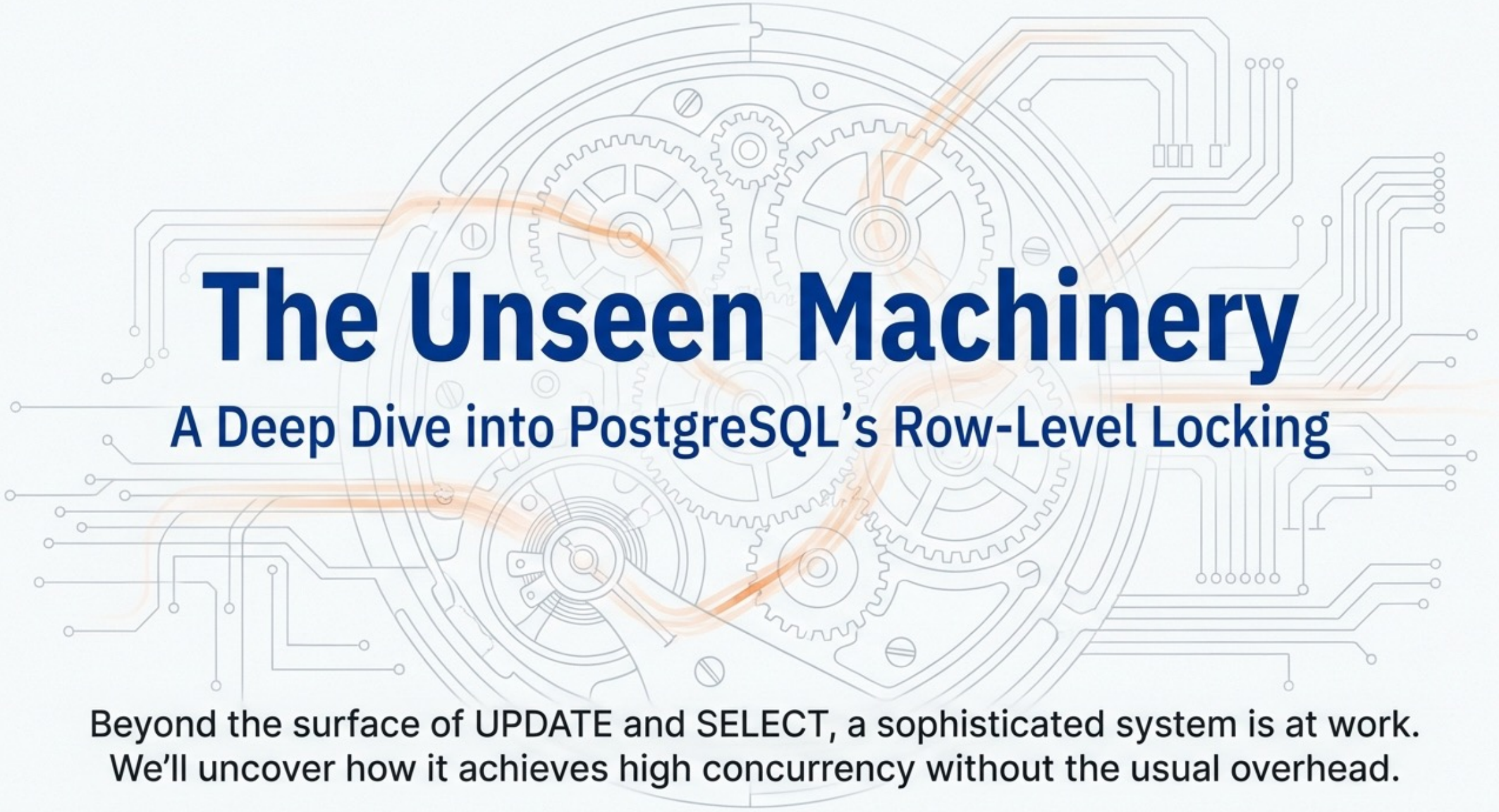


4. Your Sheet Music

Use `pg_locks` and `pg_blocking_pids()` to observe the locking symphony in real-time and diagnose contention issues.

NotebookLM

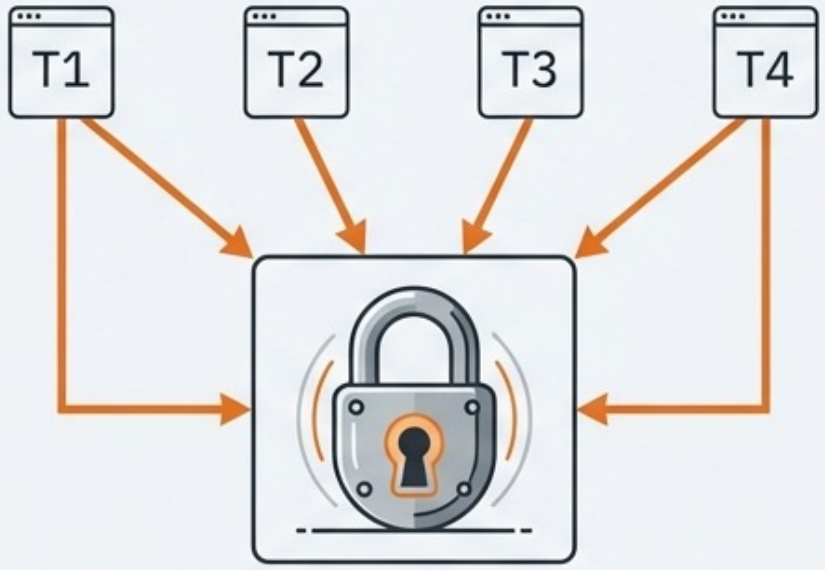
13. Row-Level Locks


The Unseen Machinery
A Deep Dive into PostgreSQL's Row-Level Locking
Beyond the surface of UPDATE and SELECT, a sophisticated system is at work.
We'll uncover how it achieves high concurrency without the usual overhead.

NotebookLM

The Concurrency Challenge: Heavyweight Locks Don’t Scale

The Traditional Approach



Central Lock Manager

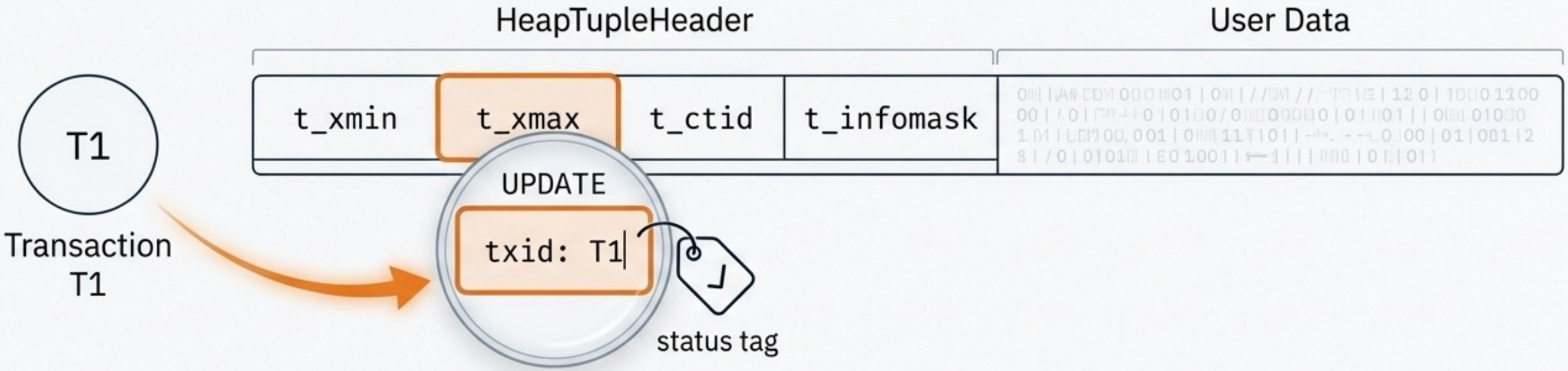
- Each lock is a heavy object, consuming significant shared memory (hundreds of bytes each).
- Managing thousands of these objects creates contention on the lock manager itself.
- Leads to ‘Lock Escalation’ (row → page → table), which kills concurrency.

The Engineering Question

How can we achieve granular, **row-level** locking for **millions** of without this massive massive overhead?

NotebookLM

PostgreSQL's Answer: A Lock Isn't an Object, It's a Status on the Row



No Central Lock Manager: Lock information is stored directly in the tuple header.

Virtually Free: Locking a row has minimal overhead—it's just a write to the `xmax` field.

Infinitely Scalable: Lock as many rows as you need without consuming RAM for locks.

This is possible thanks to PostgreSQL's MVCC architecture. Since readers don't block writers, reads don't need locks.

NotebookLM

The Language of Locks: Modes & Compatibility

The Four Row Lock Modes

Mode	Purpose
Exclusive: `UPDATE` (key can change), `NO KEY UPDATE` (key cannot change).	*Modifying or deleting a row.*
Shared: `SHARE` (prevents any changes), `KEY SHARE` (prevents key changes).	*Foreign key integrity checks.*

The Compatibility Matrix

	Key Share	Share	No Key Update	Update
Key Share				✗
Share		✗	✗	✗
No Key Update		✗	✗	✗
Update	✗	✗	✗	✗

Key Insight

Notice: `Key Share` (used for Foreign Keys) is compatible with `No Key Update` (the most common `UPDATE` lock). This is a critical design choice that prevents routine FK checks from blocking most row updates.

NotebookLM

Decoding the Lock: The Tuple Header Tells the Story

Example 1: A Standard `UPDATE`

```
-- Transaction 1
BEGIN;
UPDATE accounts SET amount = amount +
  100.00 WHERE id = 1;
```

Example 2: A `SELECT...FOR UPDATE`

```
pageinspect

-- Transaction 2
BEGIN;
SELECT * FROM accounts
WHERE id = 1 FOR NO KEY UPDATE;
```

The `lock_only` bit is `false`. This row version is being replaced.

```
pageinspect

 ctid  |  xmax  | lock_only | keys_upd
-------+--------+-----------+---------
 (0,1) | 122858 |           |
 (0,2) | 122858 |           | t
```

The `lock_only` hint bit is `true`. The row version is locked but remains the current, valid version.

```
 ctid  |  xmax  | lock_only | keys_upd
-------+--------+-----------+---------
 (0,1) | 122859 |     t     |
 (0,2) | 122859 |     t     | t
```

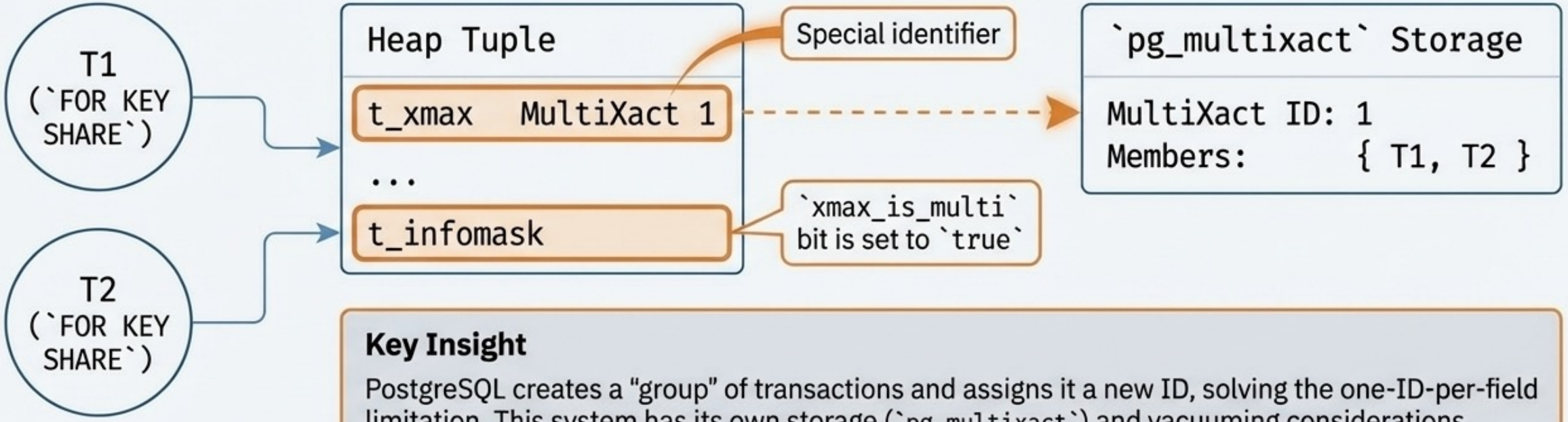
NotebookLM

Handling Shared Locks: The MultiXact Mechanism

The Problem

The `xmax` field can only hold a single transaction ID. What happens when multiple transactions need to share a lock on the same row?

The Solution: MultiXact IDs



Key Insight

PostgreSQL creates a "group" of transactions and assigns it a new ID, solving the one-ID-per-field limitation. This system has its own storage (`pg_multixact`) and vacuuming considerations.

NotebookLM

The Waiting Game: How Transactions Form a Queue



Check `xmax`

Transaction sees the row is locked by an incompatible mode.



Acquire Tuple Lock

It acquires an exclusive *heavyweight* lock on the tuple's physical identity `(page, item)`. This is the 'ticket' to get in line.



Wait on Transaction ID

It requests a lightweight lock on the `xmax` transaction ID itself and goes to sleep.



Acquire Row Lock & Release

Once woken up, it writes its own TXID to `xmax`, sets hint bits, and releases the heavyweight Tuple Lock for the next in line.

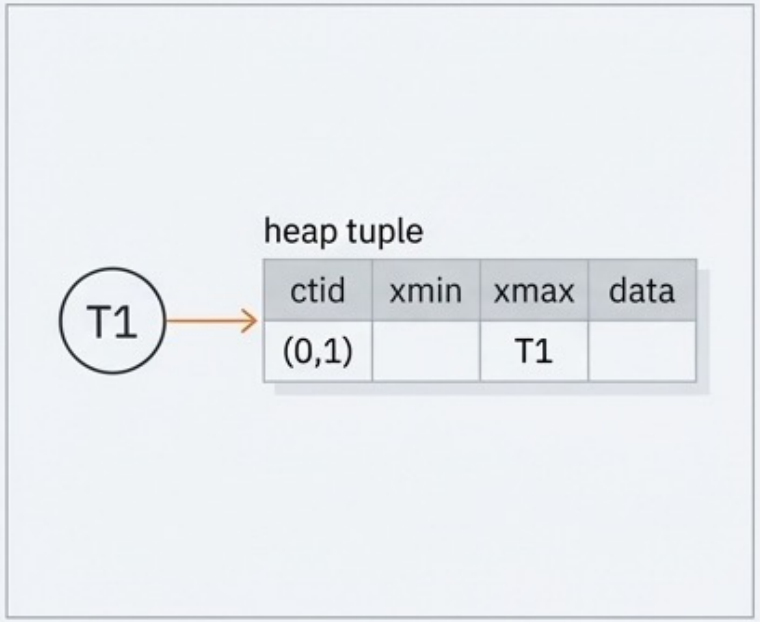
Key Insight

The heavyweight "Tuple Lock" (Step 2) is the key! It prevents a "thundering herd" and ensures an orderly First-In, First-Out queue, preventing resource starvation.

NotebookLM

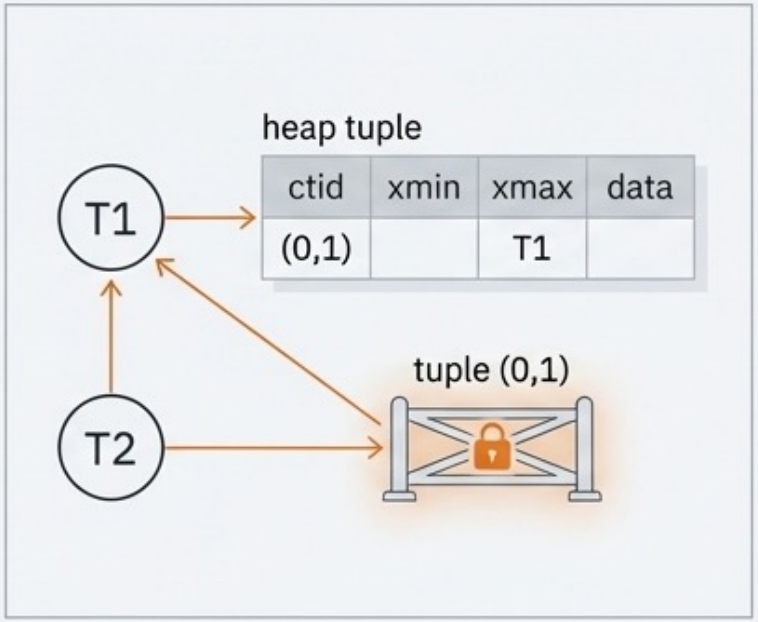
The Queue in Action: A Visual Walkthrough

T1 Holds the Lock



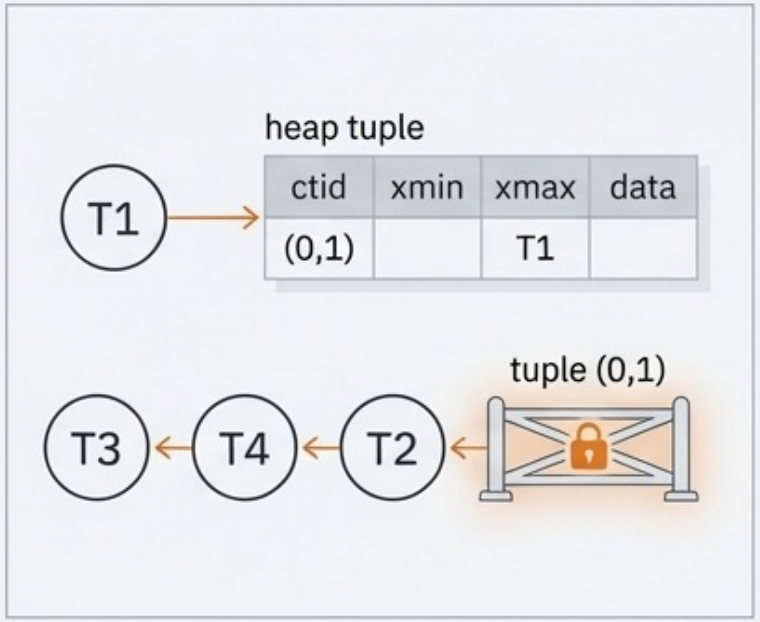
T1 holds the `xmax` lock on row (0,1).

T2 Arrives and Waits



T2 acquires the heavyweight 'tuple lock' (the gate) and waits for T1 to finish. (`wait_event: transactionid`)

T3 & T4 Queue Up



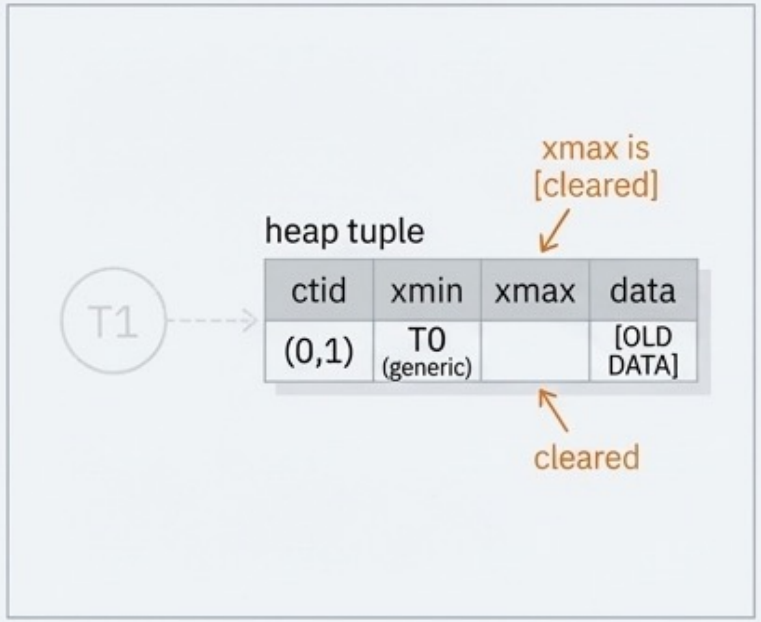
T3 and T4 are blocked trying to acquire the *tuple lock* itself, held by T2. (`wait_event: tuple`)

The tuple lock ensures an orderly FIFO queue. T2 will go next, followed by T3, then T4.

NotebookLM

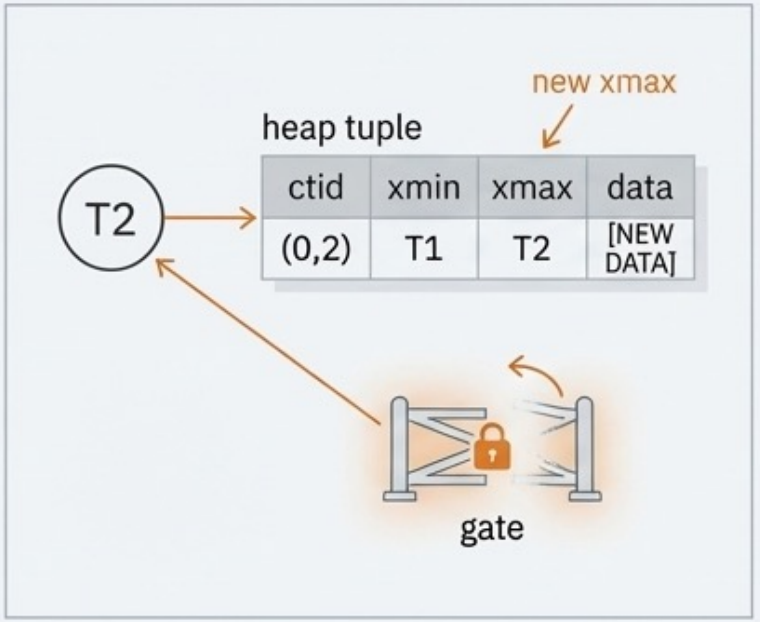
The Queue Collapses: The Danger of High-Contention Hotspots

T1 Commits



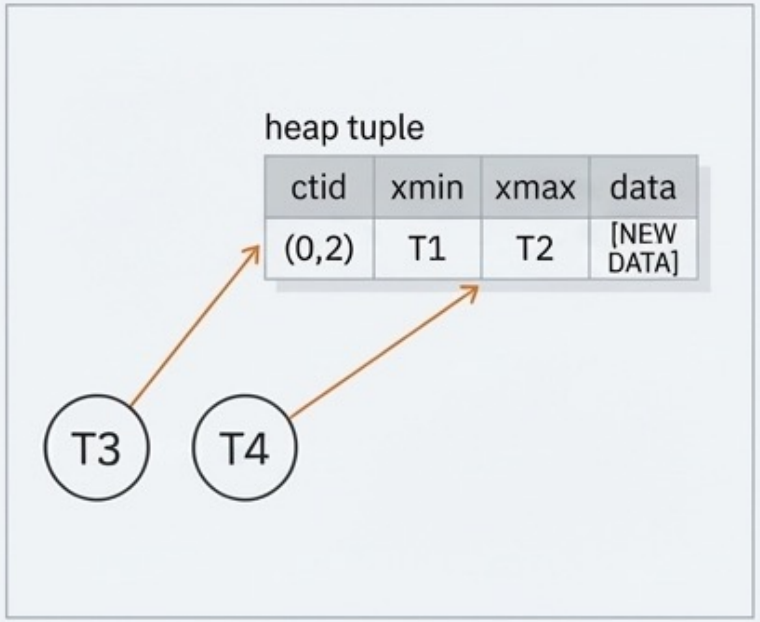
T1 commits. The original row lock is released.

T2 Proceeds



T2 wakes up, acquires the `xmax` lock, and releases its heavyweight tuple lock.

The Race Condition

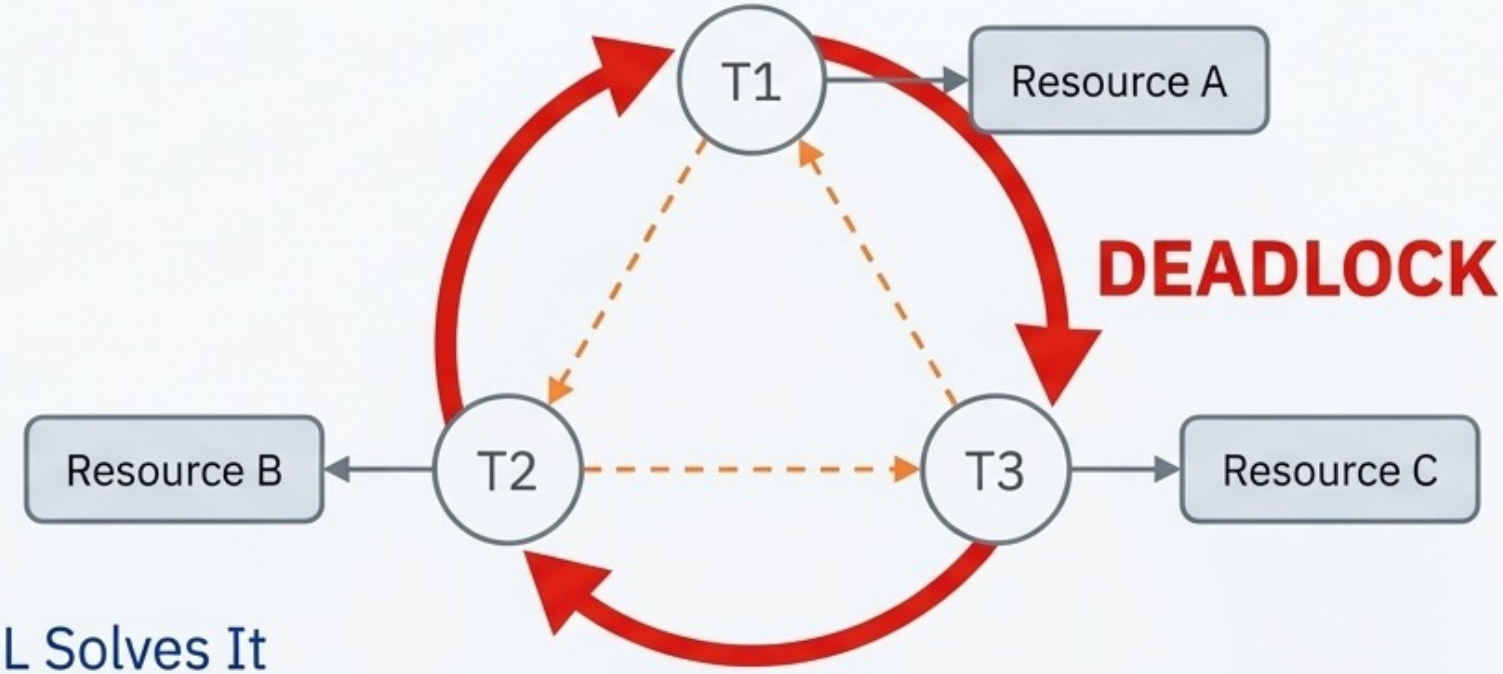


T3 and T4 wake up simultaneously. The FIFO queue is gone! They now race to wait on T2's new lock.

The orderly queue collapses after the first waiter. High-contention updates on a single row are a major performance bottleneck as the First-In, First-Out guarantee is lost.

NotebookLM

Deadlocks: The Unresolvable Circle of Waiting



How PostgreSQL Solves It

- **Detection, Not Prevention:** PostgreSQL does not try to prevent deadlocks.
- `deadlock_timeout`: After waiting (default: 1s), a process wakes up and checks for a dependency cycle in the wait-for graph.
- **The Sacrifice:** If a cycle is found, one transaction is chosen as the victim and is rolled back with an error, allowing the others to proceed.

Deadlocks are an application design problem. The primary fix is to always lock resources in a consistent, canonical order.

NotebookLM

The Hidden Deadlock: When a Single `UPDATE` Isn't Atomic

The Scenario

- Two transactions run the same conceptual update, but their execution plans cause them to process rows in a different order.
  - **Transaction A:** `UPDATE accounts SET ...` (Uses a Sequential Scan, processing rows 1, 2, 3...)
  - **Transaction B:** `UPDATE accounts SET ...` (Uses an Index Scan on a descending index, processing rows 3, 2, 1...)



The physical order in which rows are processed matters. Inconsistent execution plans under high concurrency can lead to surprising deadlocks.

NotebookLM

Controlling the Wait: `NOWAIT` and `SKIP LOCKED`



`NOWAIT`

```
SELECT ... FOR UPDATE NOWAIT;
```

Behavior: If the desired row is locked, don't wait. Immediately fail with an error: `ERROR: could not obtain lock on row...`

Use Case: Interactive applications where a user needs immediate feedback instead of a hanging UI.



`SKIP LOCKED`

```
SELECT ... FOR UPDATE SKIP LOCKED;
```

Behavior: If a row is locked, simply ignore it and move to the next available one that matches the `WHERE` clause.

Use Case: Building highly concurrent work queues. Multiple workers can pull tasks from the same table without blocking each other.

NotebookLM

Setting Boundaries: The `lock_timeout` Safety Net

```
SET lock_timeout = '1s';

ALTER TABLE accounts RENAME COLUMN amount
to balance;

-- After 1 second of waiting...
ERROR:  canceling statement due to lock
timeout
```



- Aborts any statement that waits for a lock longer than the specified time.
- Crucial for migrations (`ALTER TABLE`) on busy systems to prevent blocking all application activity.
- Can be set per-session or per-transaction for granular control.
- Distinct from `` `statement_timeout` ``, which limits total execution time. `` `lock_timeout` `` *only* applies to the time spent waiting for a lock to be granted.

NotebookLM

Mastering Concurrency: Key Principles & Takeaways



Embrace the Design

Appreciate that PostgreSQL's `xmax` locking is a feature. It's fast and scalable for the common case.



Avoid Row Hotspots

The primary cause of trouble. Avoid designs where many concurrent transactions frequently update the *same single row*.



Lock in Canonical Order

To prevent deadlocks, ensure all transactions acquire locks on multiple resources in a consistent order (e.g., always by ascending primary key).



Use the Right Tools

Leverage `SKIP LOCKED` for building efficient work queues and `lock_timeout` as a safety valve for critical operations.

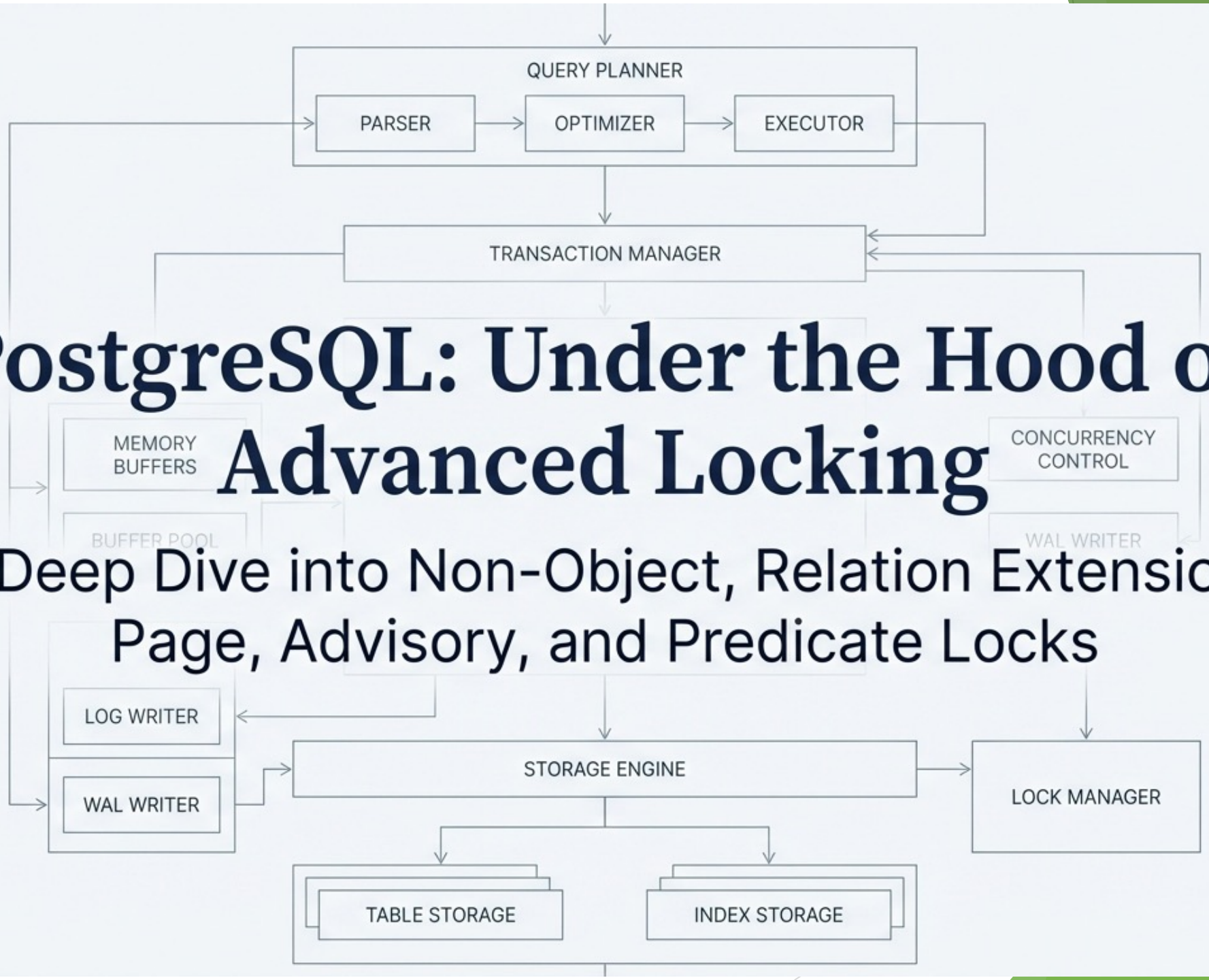


Monitor Your System

Keep an eye on `pg_locks` (for contention) and `pg_stat_database` (for deadlocks) to understand your application's real-world locking behavior.

NotebookLM

14. Miscellaneous Locks



PostgreSQL: Under the Hood of Advanced Locking

A Deep Dive into Non-Object, Relation Extension, Page, Advisory, and Predicate Locks

NotebookLM

Beyond `ROW` and `TABLE`: The Unseen Guardians of Consistency

This presentation explores the special-purpose locks that operate behind the scenes. We will peel back the layers of the engine, moving from the physical to the abstract to build a complete mental model.

3. The Virtual Layer

Guaranteeing serializability. Understanding the markers that prevent subtle data anomalies in the strictest isolation level.

(Covers: Predicate Locks)

2. The Logical Layer

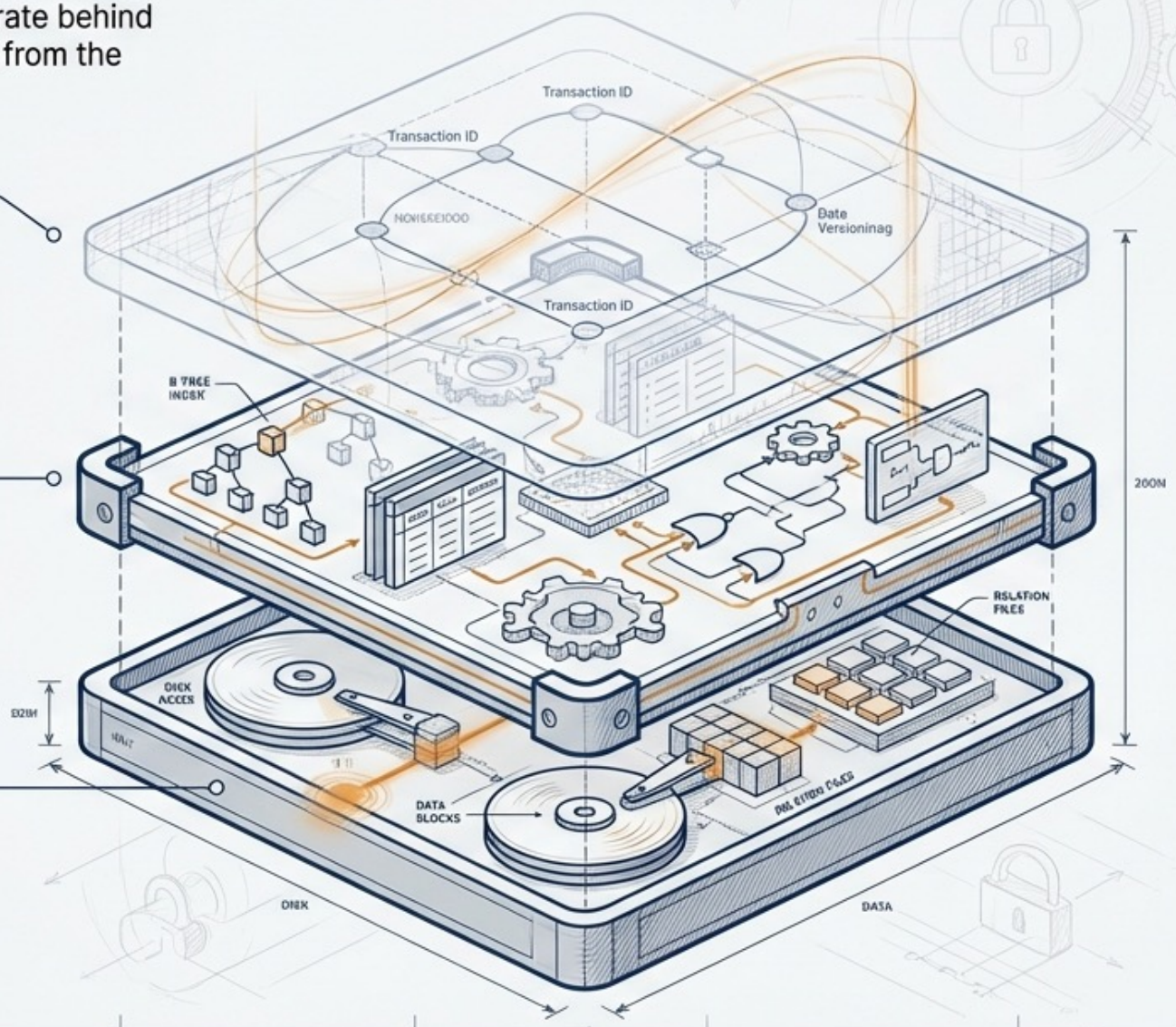
Protecting metadata and applications. Locks that coordinate access to database structure and enable user-defined logic.

(Covers: Non-Object, Advisory Locks)

1. The Physical Layer

Managing data on disk. How PostgreSQL handles the physical growth and structure of data files.

(Covers: Relation Extension, Page Locks)



NotebookLM

The Growing Pains Lock: Safely Extending Tables and Indexes

The Scenario

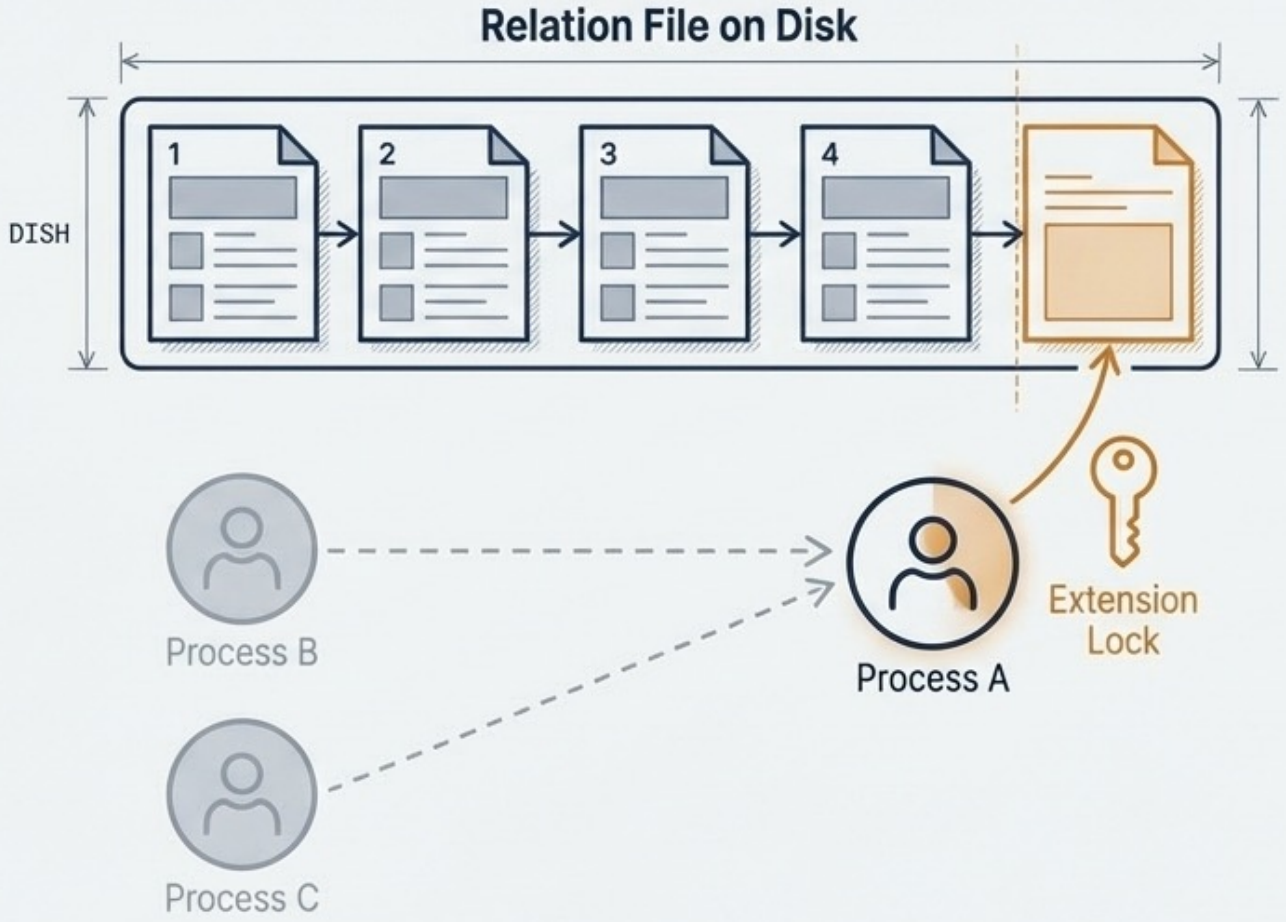
What happens when multiple processes try to insert data into a table or index simultaneously, and there's no free space? To prevent corruption, the underlying file must grow, and that operation needs to be serialized.

The Mechanism

PostgreSQL uses a special, short-lived `ExclusiveLock` on the *right to extend* the relation. This isn't a lock on the data itself, but on the metadata operation of appending a new page.

Key Characteristics

- **Short-Lived:** Released immediately after extension, not at the end of the transaction.
- **No Deadlocks (Usually):** Because it's released so quickly, it typically doesn't participate in the deadlock detection graph.
- **Optimized:** To minimize contention, heaps are extended in batches (proportional to waiting processes, up to 512 pages), while B-Tree indexes extend one page at a time.



NotebookLM

The High-Speed Merge: A Specialized Lock for GIN Indexes

The Scenario

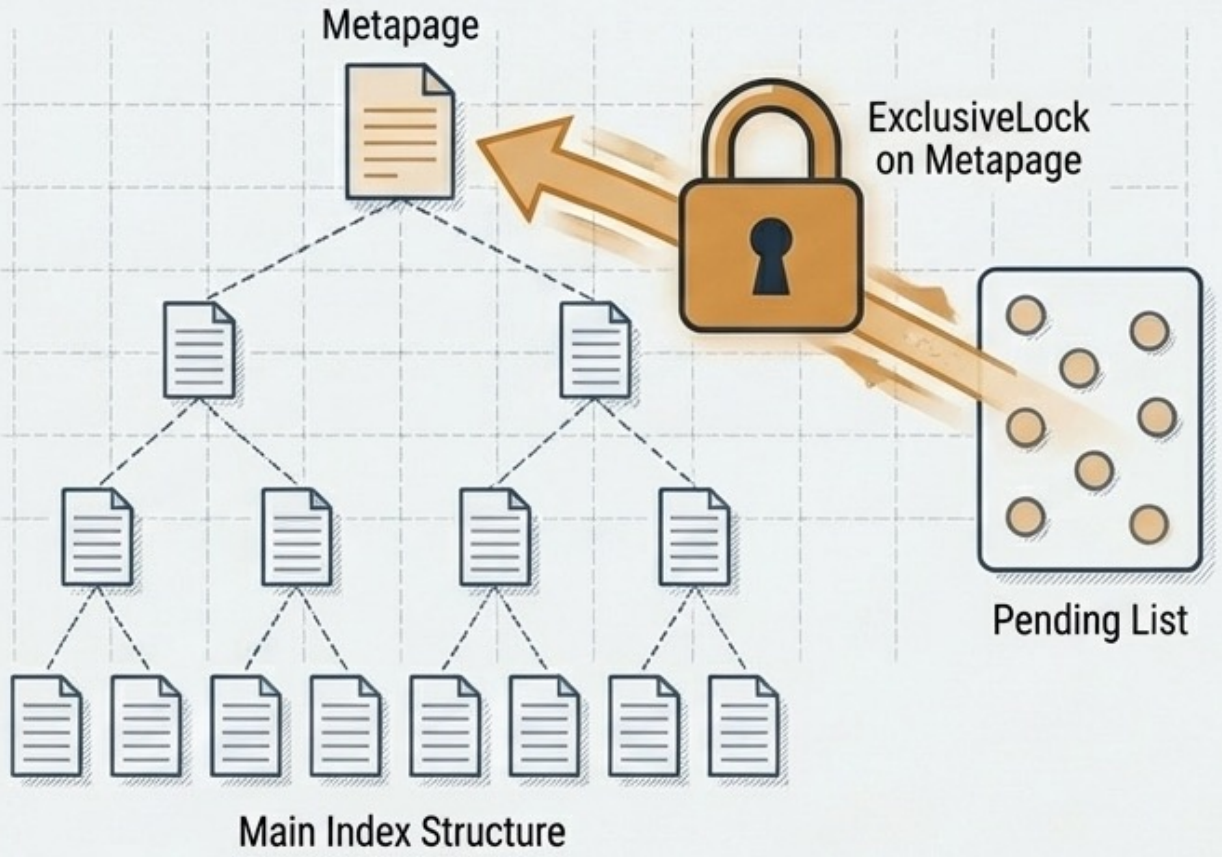
GIN indexes achieve fast writes by using a "pending list" for new entries, a feature controlled by `fastupdate=on` (the default). Periodically, these entries must be efficiently moved from the unstructured pending list into the main B-Tree-like index structure. How is this merge operation coordinated?

The Mechanism

An `ExclusiveLock is taken on the index's metapage. This lock specifically protects the operation of transferring entries from the pending list to the main index, preventing multiple processes from doing it at once. Normal index reads and writes to the pending list are not blocked.

Key Characteristics

- **GIN-Specific**: Only used for GIN indexes with `fastupdate` enabled.
- **Operation-Scoped**: Like the extension lock, it's released as soon as the merge is complete, not at transaction end.
- **Deadlock-Free**: Does not cause deadlocks.



NotebookLM

The Metadata Guardian: Protecting the Structure of Your Database

The Scenario

Transaction A begins and runs `CREATE TABLE public.example(...)`. Before it commits, Transaction B tries to `DROP SCHEMA public`.

How does PostgreSQL prevent the schema from disappearing while it's being used?

The Mechanism

PostgreSQL uses lightweight `object` locks on system catalog entries. These locks target resources like schemas, tablespaces, roles, and more, which are not considered standard relations.

Proof from `pg_locks`

```
-- Active Transaction: CREATE TABLE public.example(...);

-- Query: SELECT locktype, classid, objid, mode
          FROM pg_locks
          WHERE locktype = 'object' AND pid = pg_backend_pid();

-[ RECORD 1 ]-----------------
locktype  | object
classid   | 2615
objid     | 2200
mode      | AccessShareLock
```

System Catalog: `pg_namespace`

Object ID: `public` schema

Result: The `public` schema is locked, preventing concurrent DROP operations until the transaction ends.

NotebookLM

The Handshake Protocol: User-Defined, Application-Level Locking

The Concept: Your application needs to coordinate processes around a resource that isn't a database row or table. Examples include ensuring only one worker process attempts to process a specific file from an S3 bucket, or implementing a singleton background service.

The Mechanism: Advisory locks are an explicit, cooperative locking system managed entirely by the application developer. They are acquired by calling specific functions and are not tied to any table or data. The resource is identified by a numeric ID, often derived from a unique name using `hashtext()`.

Key Characteristics



Cooperative

They only work if all interacting application processes agree to use them. The database enforces the lock, but not its use.



Manual Control

Developers are responsible for acquiring (`pg_advisory_lock`), trying (`pg_advisory_try_lock`), and releasing (`pg_advisory_unlock`) the locks.



Flexible Scope

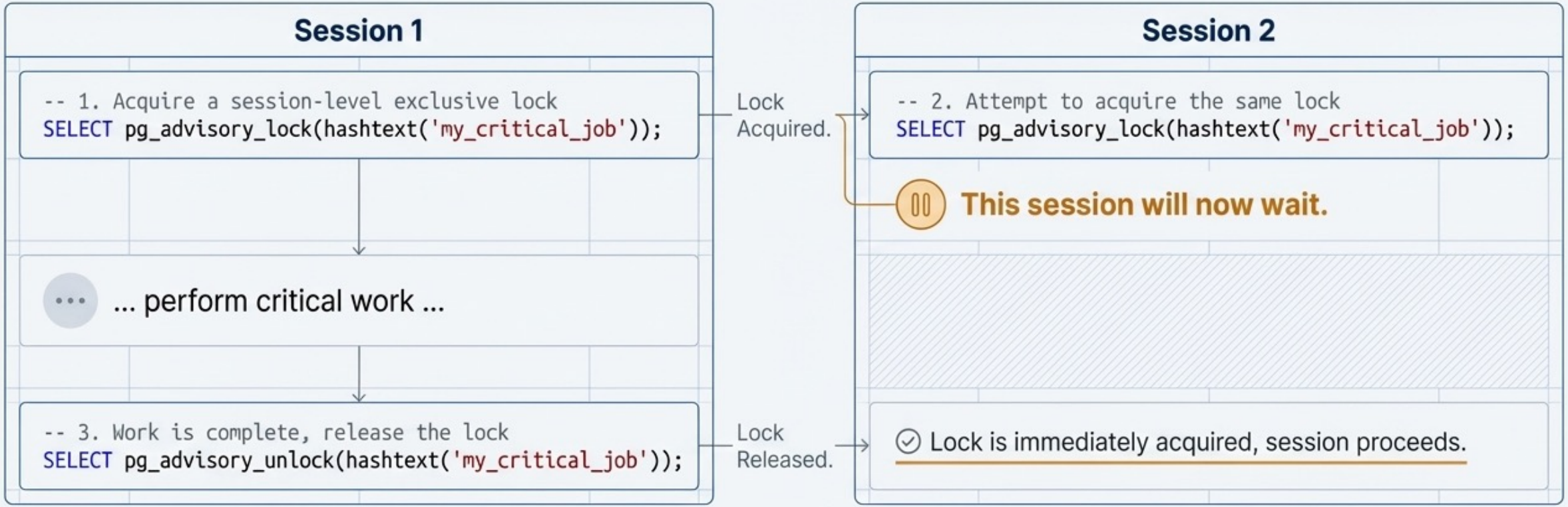
Can be scoped to a **session** (default, survives transactions) or a **transaction** (released on `COMMIT`/`ROLLBACK`, using `_xact` suffixed functions).

Can be acquired in shared or exclusive mode.

NotebookLM

Advisory Locks in Action: A Practical Example

Let's walk through a typical sequence for a resource named **"my_critical_job"**. The lock ID is generated via **hashtext("my_critical_job")**.

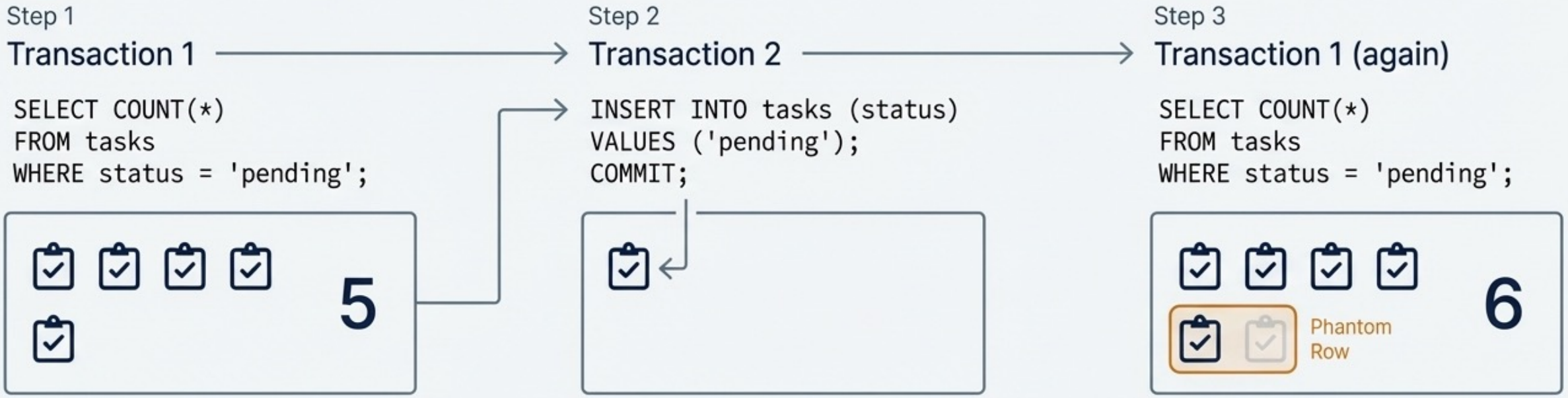


Key Takeaway

PostgreSQL provides the robust locking primitive. The application is responsible for defining the protocol and ensuring it's followed correctly.

NotebookLM

The Phantom Menace: Understanding the Phantom Read Anomaly



The Original Theory (and why it was abandoned):

The initial academic idea was to "lock the predicate" itself.

A transaction could place a lock on the condition `WHERE status = 'pending'`. Any other transaction trying to insert a row that satisfied this predicate would be blocked.

The Reality:

This is algorithmically infeasible in the general case.

Checking for predicate overlaps (e.g., between `status = 'pending''` and `priority > 10`) is computationally explosive.

The original idea of a "lock on a condition" was not implemented.

NotebookLM

A New Name for a New Job: Tracking Dependencies for SSI

The Real Purpose

In PostgreSQL's `SERIALIZABLE` isolation, these are **not locks**. They are lightweight **markers** used to track data dependencies between transactions. They don't block other transactions; they record what was read.

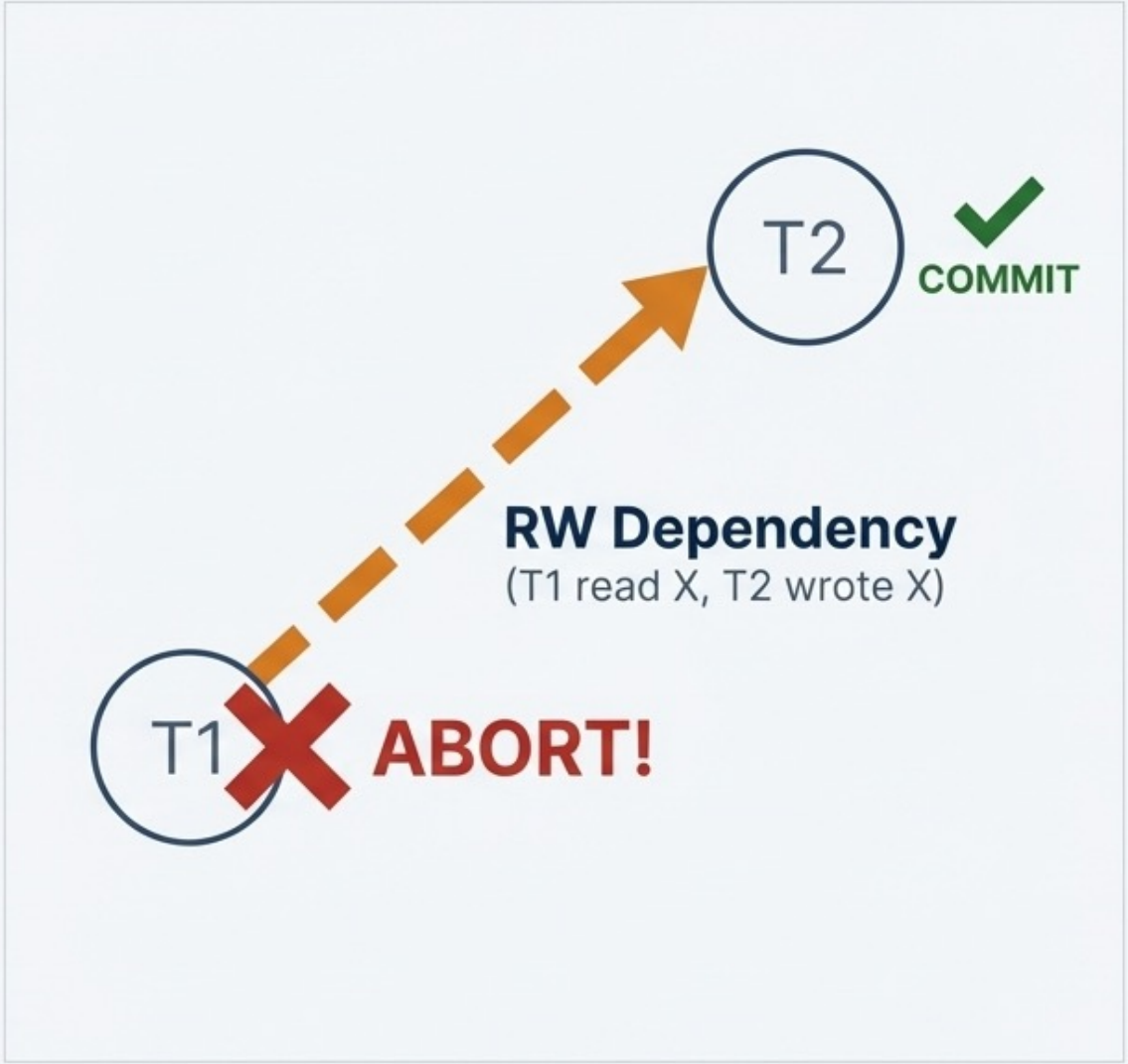
The Core Task

The SSI engine builds a dependency graph to detect "dangerous structures" that could lead to serialization anomalies. The most critical one to track is the **Read-Write (RW) dependency**:

- Transaction T1 **reads** an item X.
- Transaction T2 later **writes** to item X and commits.

The Consequence

These markers allow the engine to detect a potential anomaly. If a dangerous cycle is detected in the graph when a transaction tries to commit, it is not blocked—it is **aborted** with a serialization failure error, forcing the application to retry.



NotebookLM

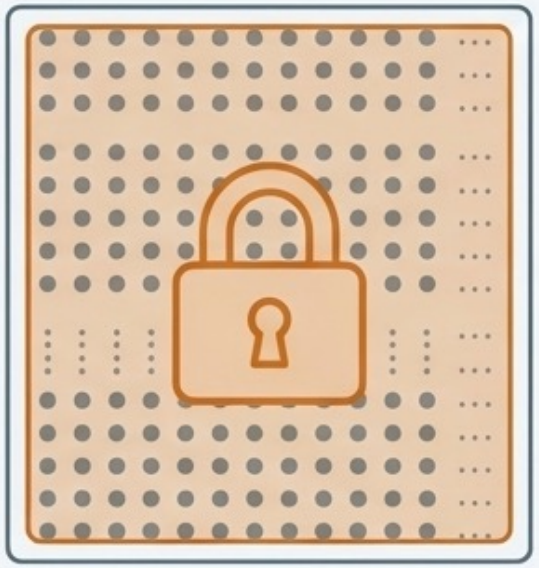
From Theory to pg_locks: Identifying the SIReadLock

When a transaction at SERIALIZABLE level reads data, it places SIReadLock markers on the objects it has touched to record the dependency. The granularity of these markers directly impacts performance and the likelihood of conflicts.

Coarse-Grained Lock

Sequential Scan

```
SELECT * FROM pred WHERE n > 100;
```



pred table

pg_locks Output

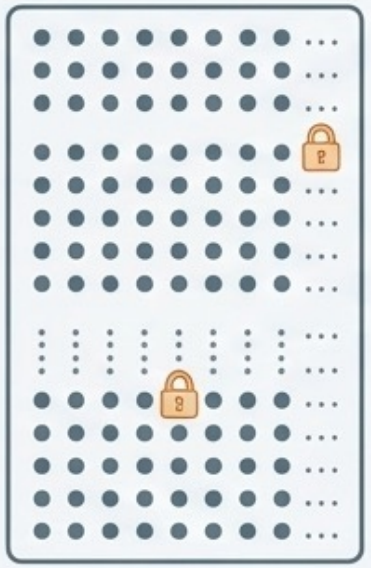
```
relation | locktype | page | tuple
---------+----------+------+------
pred     | relation |      |
```

A full table scan places a single, coarse SIReadLock on the entire relation. Any write to the table will conflict.

Fine-Grained Locks

Index Scan

```
SELECT * FROM pred WHERE n BETWEEN 1000 AND 1001;
```



pred table



pred_n_idx

pg_locks Output

```
relation   | locktype | page | tuple
-----------+----------+------+------
pred       | tuple    | 4    |    96
pred       | tuple    | 4    |    97
pred_n_idx | page     | 28   |
```

An index scan is far more precise, placing granular SIReadLocks on individual tuples read and the index pages scanned. This dramatically reduces the chance of conflict.

NotebookLM

Managing Memory: The Predicate Lock Escalation Ladder

The Problem

Tracking SIReadLock markers for thousands or millions of individual tuples would consume an unsustainable amount of shared memory.

The Solution

To conserve memory, PostgreSQL automatically promotes many fine-grained locks into a single **coarser one**. This process follows a clear hierarchy.



Level 3: Relation Lock

If the number of page/tuple locks for a single table exceeds the limit, they are all replaced by a single SIReadLock on the relation.

↑ Exceeds `max_pred_locks_per_relation` (default: 32)

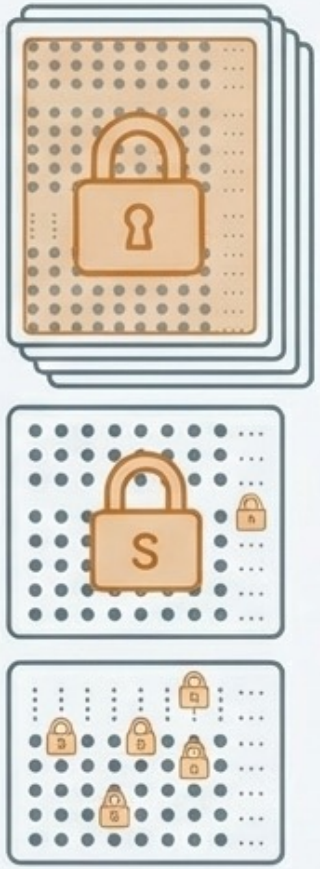
Level 2: Page Lock

If the number of tuple locks on a single data page is exceeded, they are all replaced by a single **SIReadLock** on the page.

↑ Exceeds `max_pred_locks_per_page` (default: 2)

Level 1: Tuple Locks

The system starts by taking **SIReadLocks** on individual tuples as they are read.



The Tradeoff

Escalation saves memory but increases the chance of "false positive" serialization failures. A relation-level lock means any write to that table by another serializable transaction will create a conflict, even if it doesn't touch the rows that were actually read. This can reduce throughput.

NotebookLM

Working Efficiently with `SERIALIZABLE` Isolation



1. Use Precise Index Scans

This is the most critical practice. Avoid sequential scans on large tables whenever possible. Precise queries result in fewer, more granular predicate locks, significantly reducing the probability of conflicts and serialization failures.



2. Declare `READ ONLY` Transactions

If a transaction will not modify data, explicitly start it with `BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE READ ONLY;`. This allows the lock manager to perform optimizations and potentially avoid taking predicate locks altogether.



3. Tune Memory Parameters

Be aware of the `max_pred_locks_per_transaction`, `_per_page`, and `_per_relation` settings. For workloads with large `SERIALIZABLE` transactions, you may need to increase these to avoid excessive escalation, balancing memory usage against throughput.



4. Commit to Serializability

For SSI to provide its guarantee, all transactions that access a shared set of data must use the `SERIALIZABLE` isolation level. A single transaction at a weaker level will not create `SIReadLock`s, rendering it invisible to the SSI engine.

NotebookLM

The Advanced Locking Toolkit at a Glance

Lock Type	Primary Purpose	Scope	Control	Key Characteristic
Non-Object	Protect database metadata (schemas, roles)	Transaction	Automatic	Locks catalog entries, not user data.
Relation Ext.	Safely grow table/index files on disk	Operation	Automatic	Released immediately; does not wait for transaction.
Page (GIN)	Optimize GIN index pending list maintenance	Operation	Automatic	Highly specialized, fast, and deadlock-free.
Advisory	Application-level coordination of any resource	Session/Txn	Manual	Cooperative; logic is managed by the application.
Predicate	Detect potential anomalies for `SERIALIZABLE` txn	Transaction	Automatic	A marker, not a true lock; detects RW dependencies.

NotebookLM

15. Locks on Memory Structures

Under the Hood: Decoding PostgreSQL's High-Performance Locking

A journey from wait events to CPU cycles, revealing how PostgreSQL protects shared memory and what it means for performance.

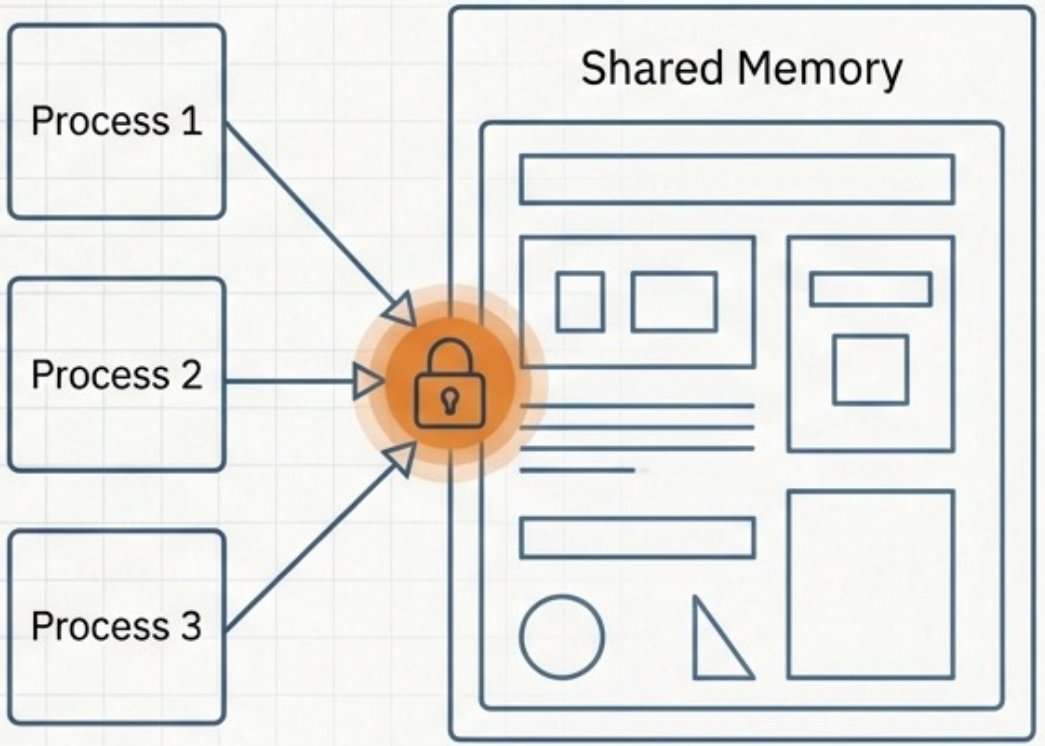
NotebookLM

The Core Challenge: Speed Through Shared Memory

To achieve maximum performance, PostgreSQL processes don’t work in isolation; they access a common pool of data structures in shared memory.

This shared access requires a sophisticated protection system to prevent data corruption from simultaneous updates.

This system is built on locks. When processes compete for these locks, it creates “wait events”—the primary clue for diagnosing performance bottlenecks.



NotebookLM

Making Waits Visible with `pg_stat_activity`

The `pg_stat_activity` view is your first-line, real-time tool for observing what a process is waiting for. The `wait_event_type` and `wait_event` columns reveal the exact nature of any delay.

```
SELECT backend_type, wait_event_type AS event_type, wait_event
FROM pg_stat_activity;
```

```
         backend_type          | event_type|      wait_event
-------------------------------+-----------+----------------------
 logical replication launcher | Activity  | LogicalLauncherMain
 autovacuum launcher          | Activity  | AutoVacuumMain
 client backend               |           |
 background writer            | Activity  | BgWriterMain
 checkpointer                 | Activity  | CheckpointerMain
 walwriter                    | Activity  | WalWriterMain
```

NotebookLM

A Taxonomy of Wait Events

Not all waits indicate a problem. They are classified into several types. For our deep-dive, we are focusing on locks protecting internal memory structures.

Lock-Related Waits



`LWLock`: Lightweight locks protecting shared memory structures.



`Lock`: Heavyweight, transaction-level locks.

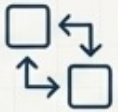


`BufferPin`: Waiting for access to a buffer.

Resource Waits



`IO`: Waiting for data to be read from or written to disk.



`IPC`: Waiting for data from another process.



`Client`: Waiting for data from the client application.

Idle / Background Waits

(Generally Normal)



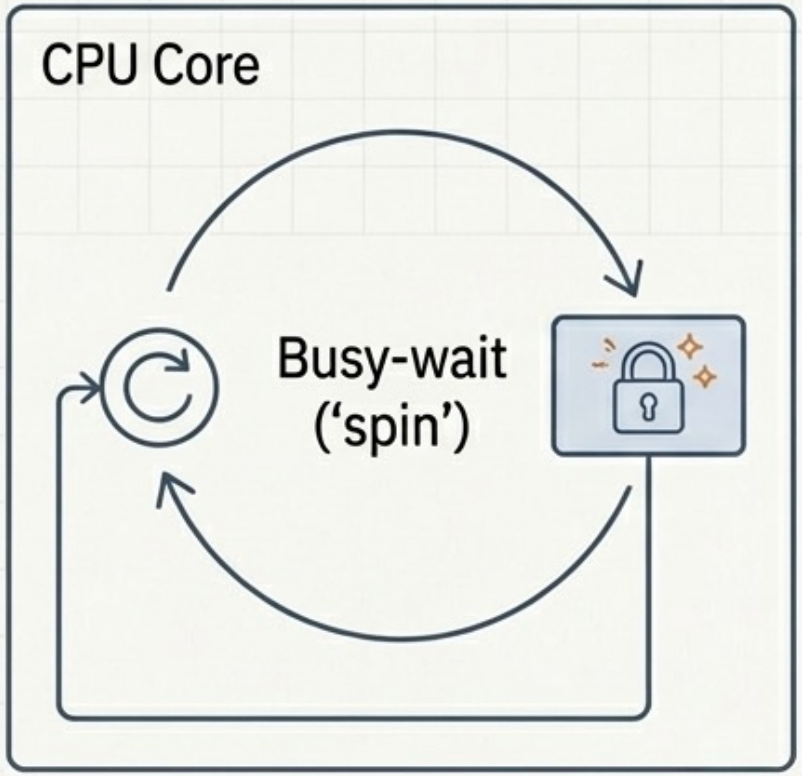
`Activity`: The main loop of a background process (e.g., `AutoVacuumMain`, `WalWriterMain`).



`Timeout`: A process is sleeping for a scheduled duration.

NotebookLM

The Foundation: Spinlocks for Microsecond Operations



A **Spinlock** is the simplest lock, designed to protect a specific memory cell for an extremely short duration, often just a few CPU cycles. If the resource is locked, the process repeatedly executes a check in a tight loop ('busy-wait').

Key Attributes:

- **Mechanism**: Atomic CPU instructions (e.g., compare-and-swap).
- **Mode**: Exclusive only.
- **Behavior**: Busy-wait ('spin'). Does not yield the CPU.
- **Visibility**: No instrumentation. They are **invisible** to monitoring tools like `pg_stat_activity`. Their correct implementation is the responsibility of PostgreSQL developers.

NotebookLM

The Workhorse: Lightweight Locks (LWLocks)

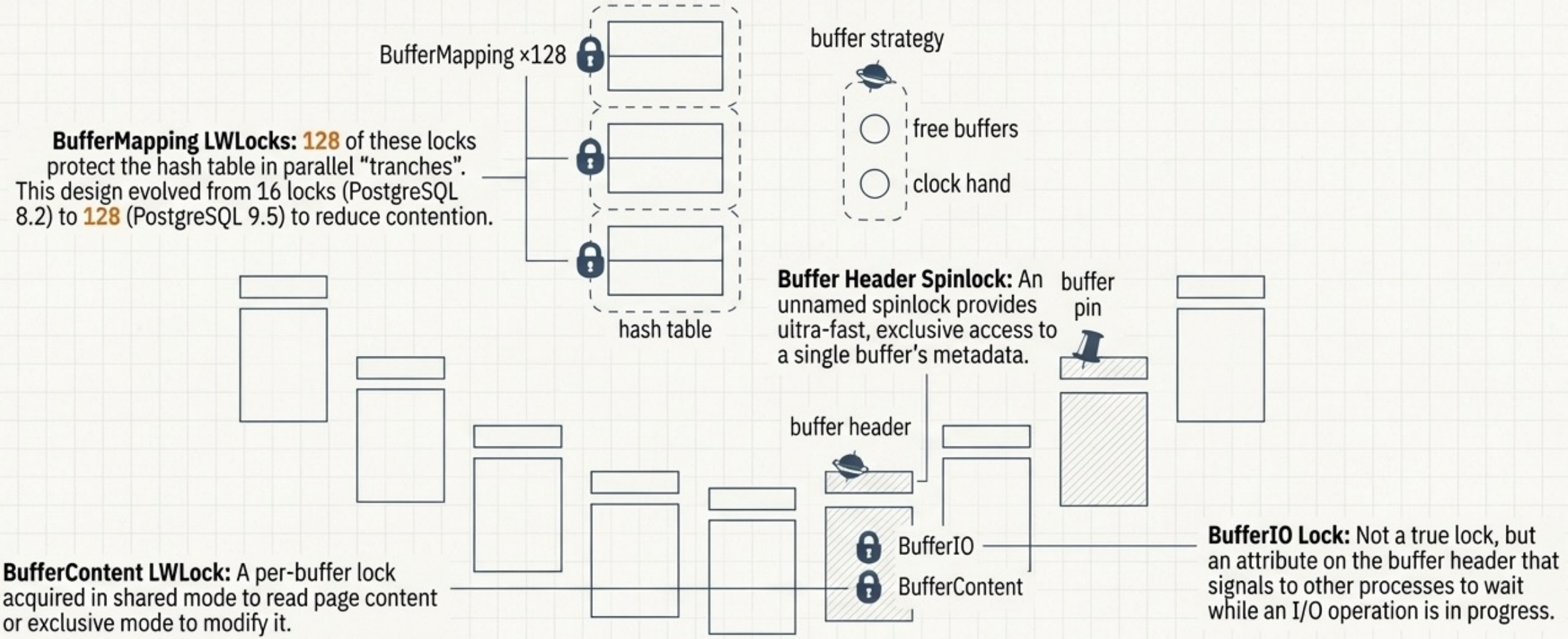
LWLocks are used to protect access to entire data structures, like a hash table. They are held for longer than Spinlocks and are the source of the `LWLock` wait events you see in monitoring.

Feature	Spinlock	Lightweight Lock (LWLock)
Purpose	Protect a single memory cell	Protect a data structure
Duration	A few CPU cycles	Short, but can cover I/O
Modes	Exclusive Only	Shared (read) & Exclusive (write)
Contention	Busy-wait (spins on CPU)	Sleeps (yields to OS)
Visibility	Not instrumented (**invisible**)	Instrumented (**visible** in `pg_stat_activity`)

NotebookLM

Case Study: A Symphony of Locks in the Buffer Cache

The Buffer Cache is a high-traffic area with a sophisticated, fine-grained locking strategy to maximize parallelism on multi-core systems.



BufferMapping LWLocks: 128 of these locks protect the hash table in parallel "tranches". This design evolved from 16 locks (PostgreSQL 8.2) to 128 (PostgreSQL 9.5) to reduce contention.

Buffer Header Spinlock: An unnamed spinlock provides ultra-fast, exclusive access to a single buffer's metadata.

BufferContent LWLock: A per-buffer lock acquired in shared mode to read page content or exclusive mode to modify it.

BufferIO Lock: Not a true lock, but an attribute on the buffer header that signals to other processes to wait while an I/O operation is in progress.

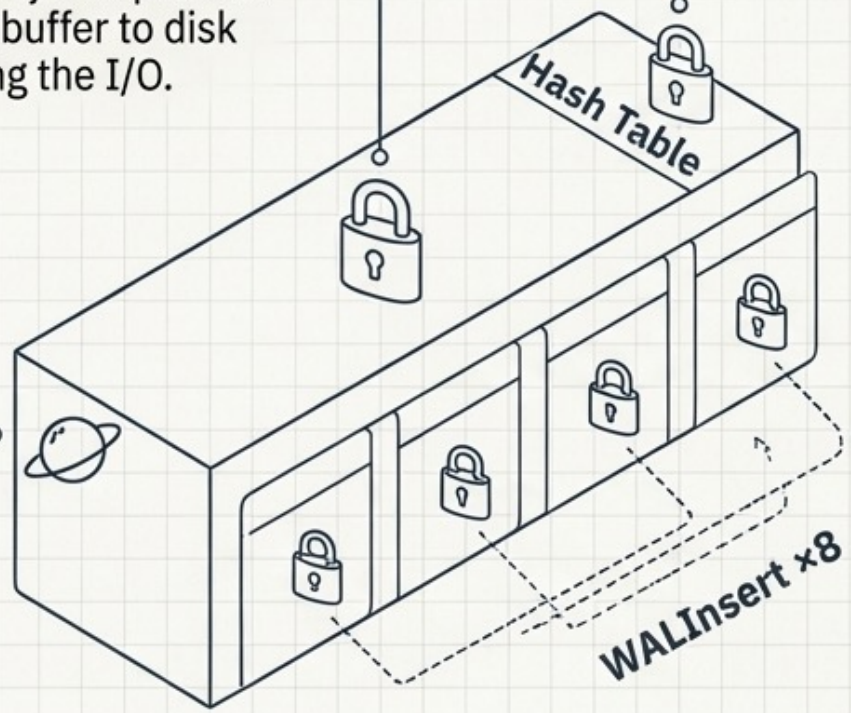
NotebookLM

Case Study: Sequential Precision in the WAL Buffer

Unlike the Buffer Cache's random access pattern, the WAL is primarily sequential. Its locking strategy is simpler and optimized for this specific workflow.

`WALWrite` LWLock: A single LWLock ensures only one process can flush the WAL buffer to disk at a time, serializing the I/O.

`WALBufMapping` LWLock: A single LWLock protects the WAL's hash table, sufficient due to its smaller size and more sequential access.



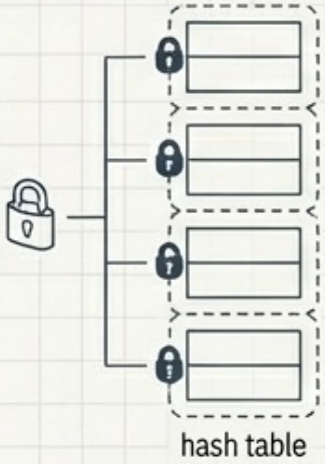
`insert position` Spinlock: A classic spinlock use case: protecting the single pointer that tracks where the next WAL record should be written.

`WALInsert` LWLocks: A tranche of 8 LWLocks allows multiple processes to *fill* their reserved space in the buffer in parallel.

NotebookLM

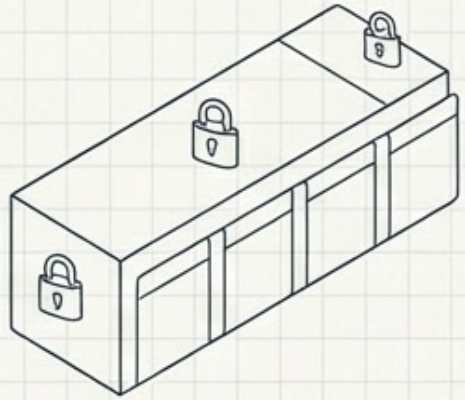
The Right Lock for the Right Job

PostgreSQL's locking is a masterclass in tailored design. The strategy changes dramatically based on the data structure's access pattern.



Buffer Cache: High Concurrency, Random Access

- Requires massively parallel access.
- **Solution:** Fine-grained locking. The hash table is split into 128 `BufferMapping` tranches to minimize contention between processes accessing different parts of the cache.

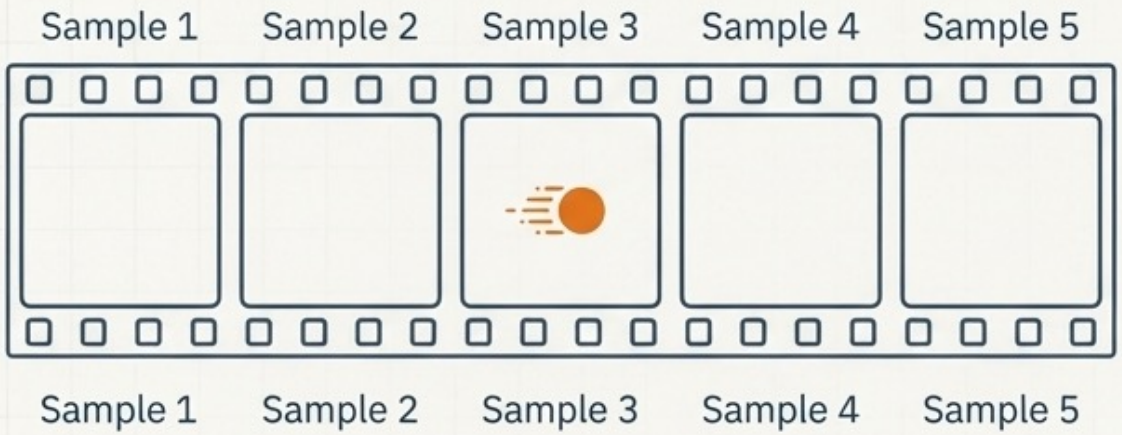


WAL Buffer: High Throughput, Sequential Access

- Requires strict ordering for writes.
- **Solution:** Targeted locking. A spinlock (`insert position`) protects the critical insertion point, while a single LWLock (`WALWrite`) serializes the final disk write.

NotebookLM

The Problem with Snapshots: A Single Moment Isn’t the Whole Story

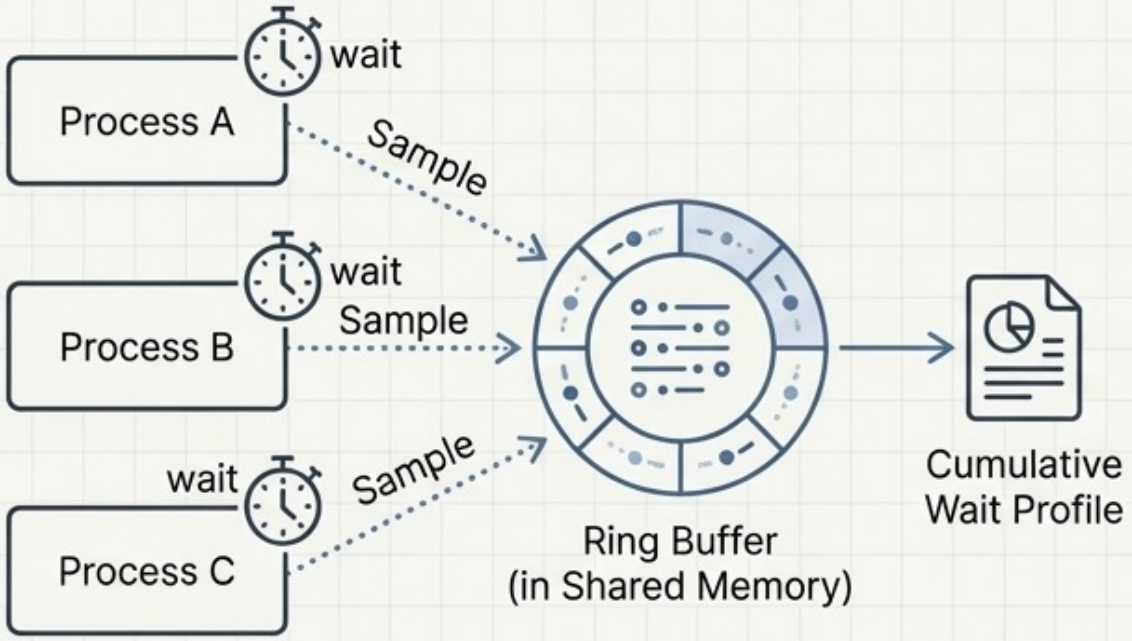


`pg_stat_activity` might only catch a wait event event if you’re lucky.

- pg_stat_activity is powerful, but it only shows the wait event happening *at the exact moment you* query it*.
- It does not accumulate statistics over time.
- Very short but frequent wait events—often the cause of major performance degradation—are easily missed between samples. What we need is a cumulative profile of waits over a period.

NotebookLM

The Solution: Building a Wait Profile with `pg_wait_sampling`



The `pg_wait_sampling` extension provides the missing piece. It periodically samples wait events for all processes and stores them in a ring buffer in shared memory.

Key Functionality: It allows you to query a cumulative profile of all wait events that occurred during a session, revealing which events consumed the most time overall.

Setup Note: Requires adding `pg_wait_sampling` to `shared_preload_libraries` and restarting the server, followed by `CREATE EXTENSION`.

NotebookLM

A `pgbench` Profile Under Normal Load

After running a standard `pgbench` test, we can query the wait profile for the benchmark process.

```
SELECT event_type, event, count
FROM pg_wait_sampling_profile
WHERE pid = <pid>
ORDER BY count DESC LIMIT 4;
```

event_type	event	count
IO	WALSync	3478
IO	WALWrite	52
Client	ClientRead	30
IO	DataFileRead	2

Interpretation

Interpretation: The overwhelming majority of wait time is spent on `WALSync`. This means the workload bottleneck is the time it takes to receive confirmation that WAL records have been physically flushed to durable storage.

NotebookLM

How Bottlenecks Reveal Themselves: The Profile Under Slow I/O

When we run the same test but artificially slow down the filesystem, the wait profile shifts dramatically, revealing the underlying contention.

event_type	event	count
IO	WALWrite	3603
LWLock	WALWrite	2095
IO	WALSync	22
IO	DataFileExtend	19

The Insight

1. The `IO` wait for `WALWrite` is now the top offender, as expected with slow disks.
2. Crucially, contention on the `LWLock` also named `WALWrite` now appears as the #2 wait.
3. **This is the key connection**: The slow I/O operation forces processes to hold the `WALWrite` LWLock for much longer, causing other processes to queue up behind it. **The lock contention is a *symptom* of the slow disk.**

NotebookLM

Your Diagnostic Path: From Symptom to Source



Observe Symptom
A process is slow. Use `pg_stat_activity`.



Identify Clue
You see frequent `LWLock` wait events.



Understand Mechanism
This wait is caused by contention on internal `LWLocks` (or spinlocks, which are invisible).



Pinpoint Location
Research the specific lock name (e.g., `WALWrite`, `BufferMapping`) to understand where in the system the contention occurs (e.g., WAL Buffer, Buffer Cache).



Build a Profile
Use `pg_wait_sampling` to confirm the historical significance of the wait and distinguish chronic bottlenecks from momentary delays.

NotebookLM

Observe, Understand, Diagnose

1

Start with `pg_stat_activity`.

It is your essential real-time dashboard. Know what your processes are doing **right now**.

2

Know Your Locks.

Understand the crucial difference between invisible, CPU-bound Spinlocks and observable, versatile LWLocks. This distinction explains what you can and cannot see.

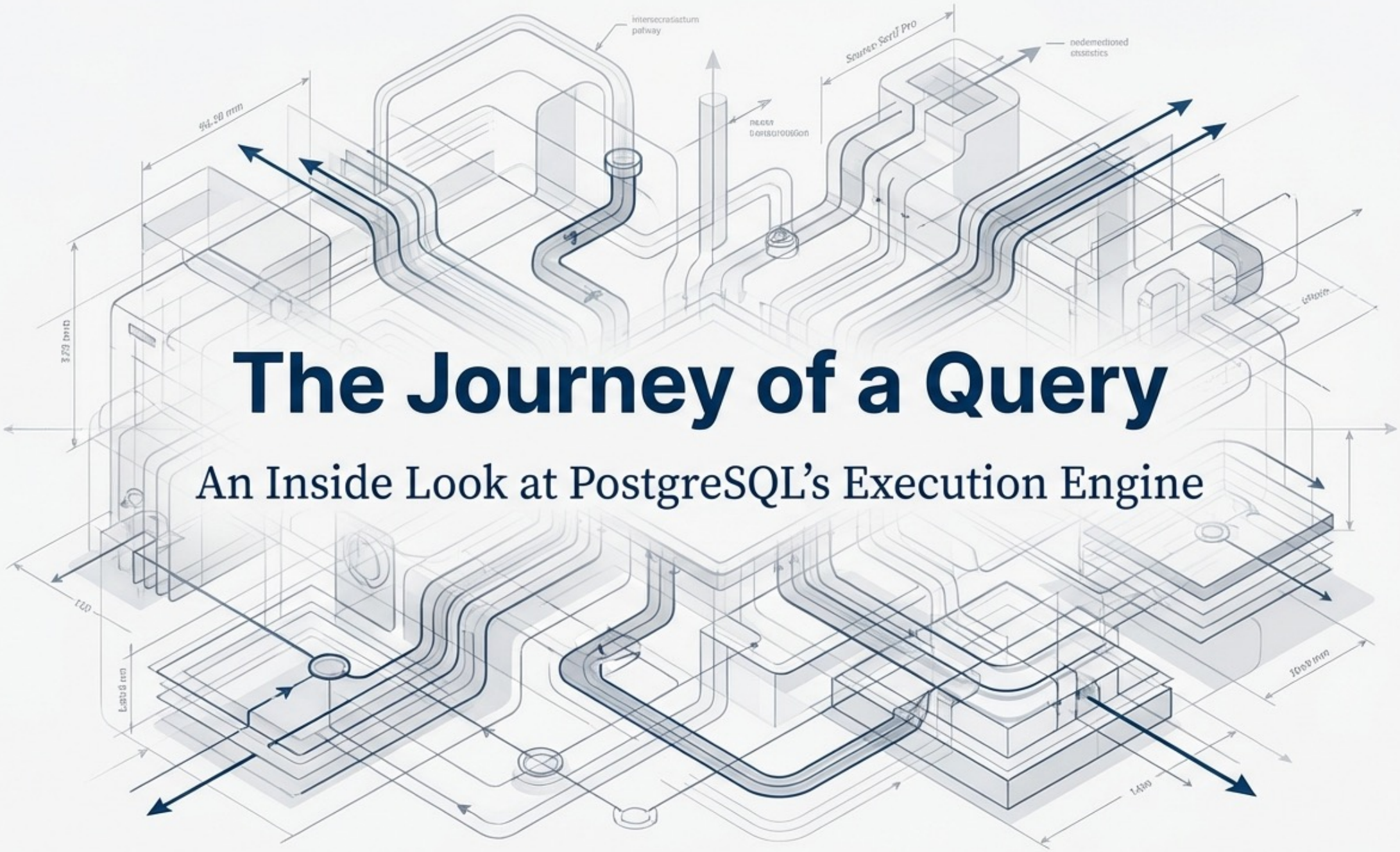
3

Profile Over Time for the Full Story.

For persistent or elusive bottlenecks, a snapshot is not enough. Use `pg_wait_sampling` to build a historical profile and uncover the true source of your wait time.

NotebookLM

16. Query Execution Stages



The Journey of a Query

An Inside Look at PostgreSQL's Execution Engine

NotebookLM

Every Query Tells a Story. This is Ours.

We will follow a single SQL query from the moment it's submitted as a simple text string to the final delivery of its results. This journey reveals the sophisticated, multi-stage process that PostgreSQL uses to understand, optimize, and execute your commands.

Our Protagonist Query:

```
SELECT schemaname, tablename
FROM pg_tables
WHERE tableowner = 'postgres'
ORDER BY tablename;
```

This query appears simple, but its journey will take us through the core components of the database engine.

NotebookLM

The Four Stages of a Query's Life

A query doesn't just 'run.' It is transformed through a pipeline of four critical stages, each with a distinct purpose.

1. Parse & Analyze

The raw text is translated into a structured, validated format the database can understand.

From text to meaning.

2. Rewrite

The query is transformed, expanding abstractions like views to reveal the underlying tables.

From meaning to a complete picture.

3. Plan

The optimizer explores countless execution paths to find the single most efficient strategy.

From picture to blueprint.

4. Execute

The chosen plan is brought to life, and the final data is retrieved and returned.

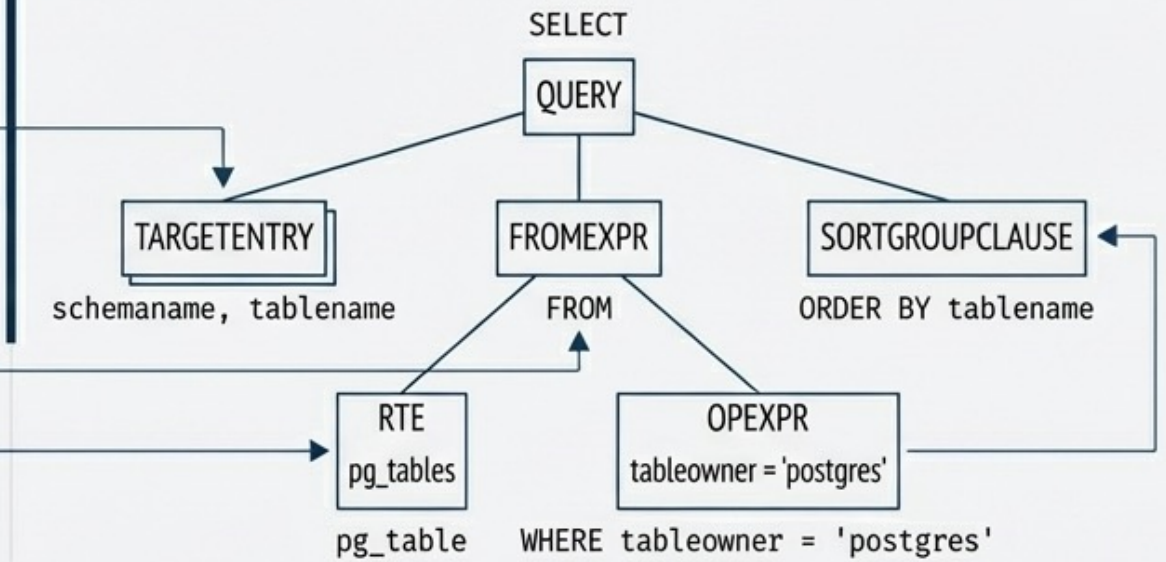
From blueprint to results.

NotebookLM

Stage 1: Parsing – From Raw Text to a Syntax Tree

The journey begins with the parser, which breaks the raw query string into recognizable tokens (*lexical analysis*) and validates it against SQL grammar (*syntax analysis*). The output is an abstract syntax tree (AST), a structured representation of the query's logic. During semantic analysis, the system verifies that objects like `pg_tables` exist and you have permission to access them.

```
SELECT schemaname, tablename
FROM pg_tables
WHERE tableowner = 'postgres'
ORDER BY tablename;
```

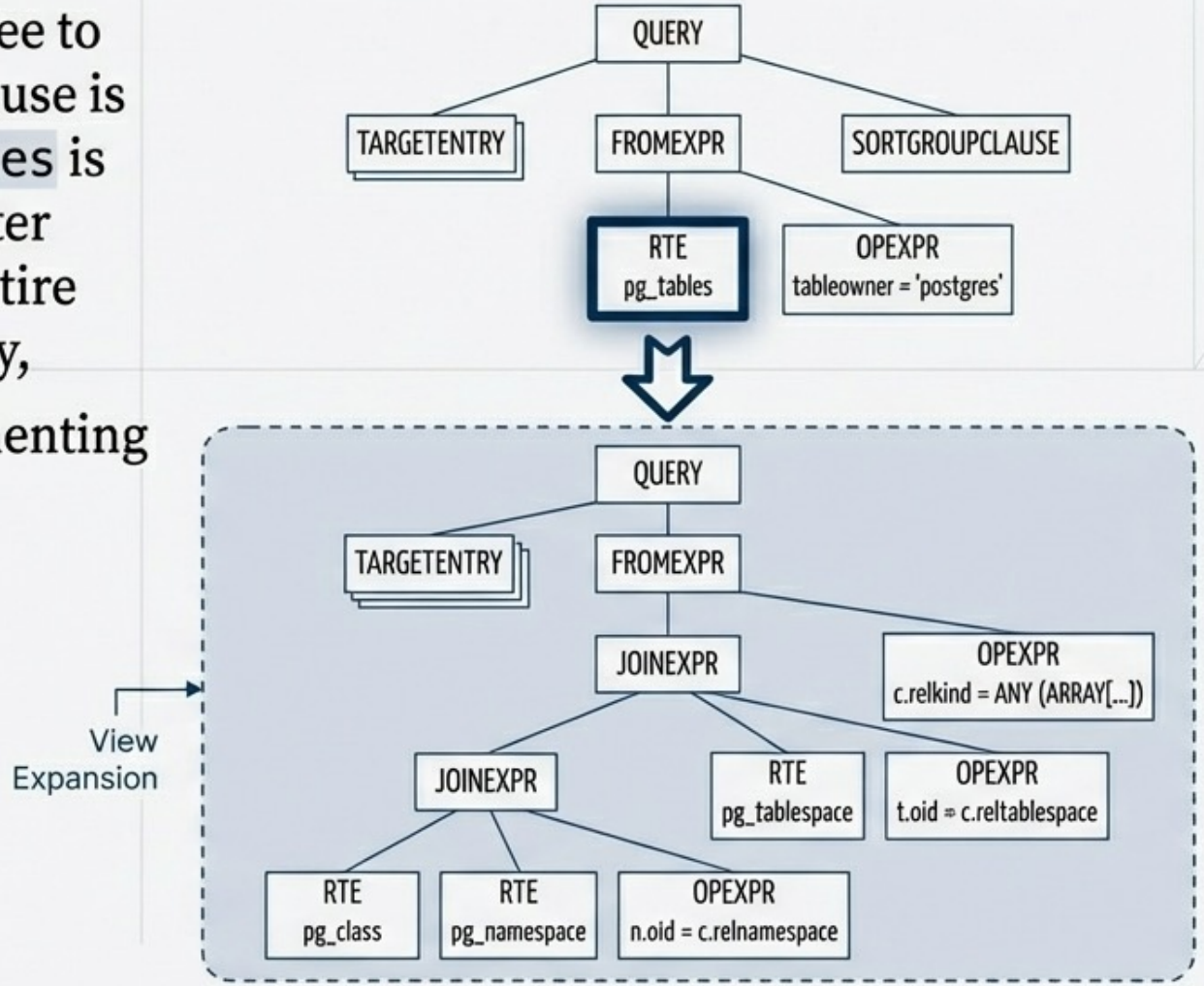


NotebookLM

Stage 2: Rewriting – Unmasking the Abstractions

The rewrite system modifies the parse tree to resolve abstractions. The most common use is expanding views. Our query on `pg_tables` is not on a real table, but a view. The rewriter replaces the `pg_tables` node with the entire representing the view's underlying query,

The same mechanism is used for implementing Row-Level Security policies.



NotebookLM

Stage 3: The Planner's Dilemma – Finding the Optimal Path

SQL specifies the desired result, not the method to retrieve it. For any non-trivial query, there are numerous possible execution paths:



- Which table should be scanned first?
- Should an index be used, or is a full table scan faster?
- Which join algorithm (Nested Loop, Hash Join, Merge Join) is best for these tables?

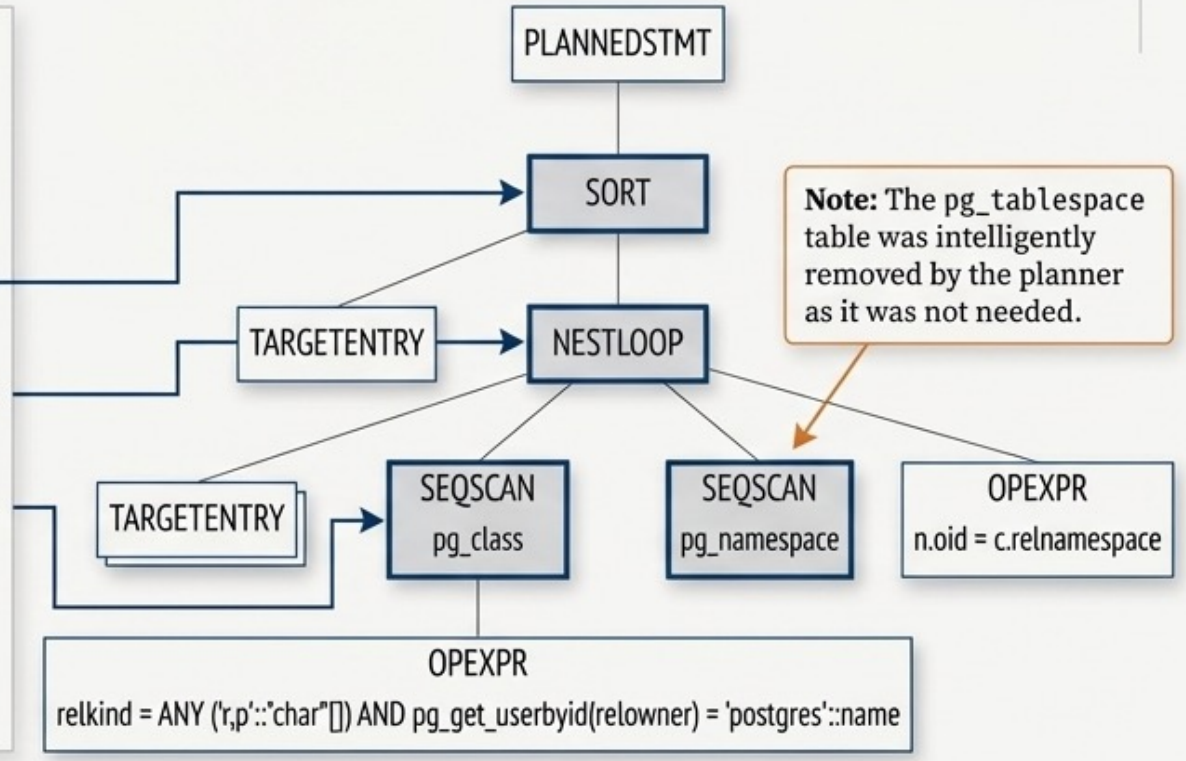
The planner, also known as the optimizer, is responsible for answering these questions. Its goal is to find the plan with the lowest estimated execution 'cost.' An optimal plan can be orders of magnitude faster than a poor one.

NotebookLM

The Blueprint for Execution: The Plan Tree

The planner's output is an execution plan, a tree of physical operations. Unlike the parse tree, which represents *what* to do, the plan tree specifies *how* to do it. Each node is an action (e.g., scan a table, perform a join, sort results).

```
EXPLAIN SELECT ...
                          QUERY PLAN
-----------------------------------------------------------------
Sort  (cost=21.03..21.04 rows=1 width=128)
  Sort Key: c.relname
  -> Nested Loop Left Join  (cost=0.00..21.02 rows=1 width=128)
     -> Seq Scan on pg_class c  (cost=0.00..19.93 rows=1 ...)
     -> Seq Scan on pg_namespace n  (cost=0.00..1.04 rows=4 ...)
```



NotebookLM

How the Planner Estimates Cost: Statistics are Everything

PostgreSQL uses a cost-based optimizer. The 'cost' is an arbitrary unit representing a combination of I/O and CPU work. The planner calculates this cost for every potential plan and picks the cheapest one. The estimation process is recursive and relies on two key inputs:

Cardinality

The estimated number of rows a node will process or output.

Selectivity

The fraction of rows from an input set that will remain after a filter (e.g., a `WHERE` clause) is applied. A highly selective filter removes most rows.

Key Insight

These estimations depend almost entirely on statistical data about your tables (size, data distribution, etc.) collected by the `ANALYZE` command (often run by autovacuum). **Inaccurate or stale statistics are the most common cause of poor query plans.**

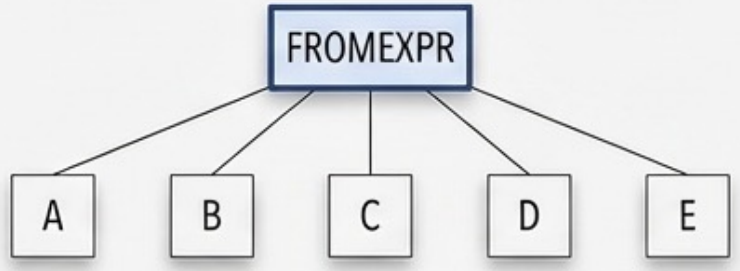
NotebookLM

Taming Complexity: Managing Join Order

The number of possible join orders grows exponentially with the number of tables, making an exhaustive search impossible for complex queries. PostgreSQL uses dynamic programming and heuristics to navigate this search space. You can influence this process:

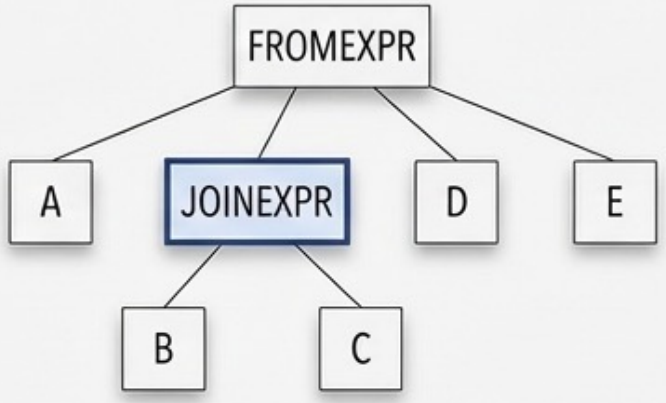
`FROM a, b, c, d, e`

Gives the planner complete freedom to reorder all tables.



`FROM a, b JOIN c ON ..., d, e`

Using explicit `JOIN` syntax provides a hint. The planner will generally avoid reordering these joins if the number of items in the `FROM' list exceeds `join_collapse_limit` (default: 8).



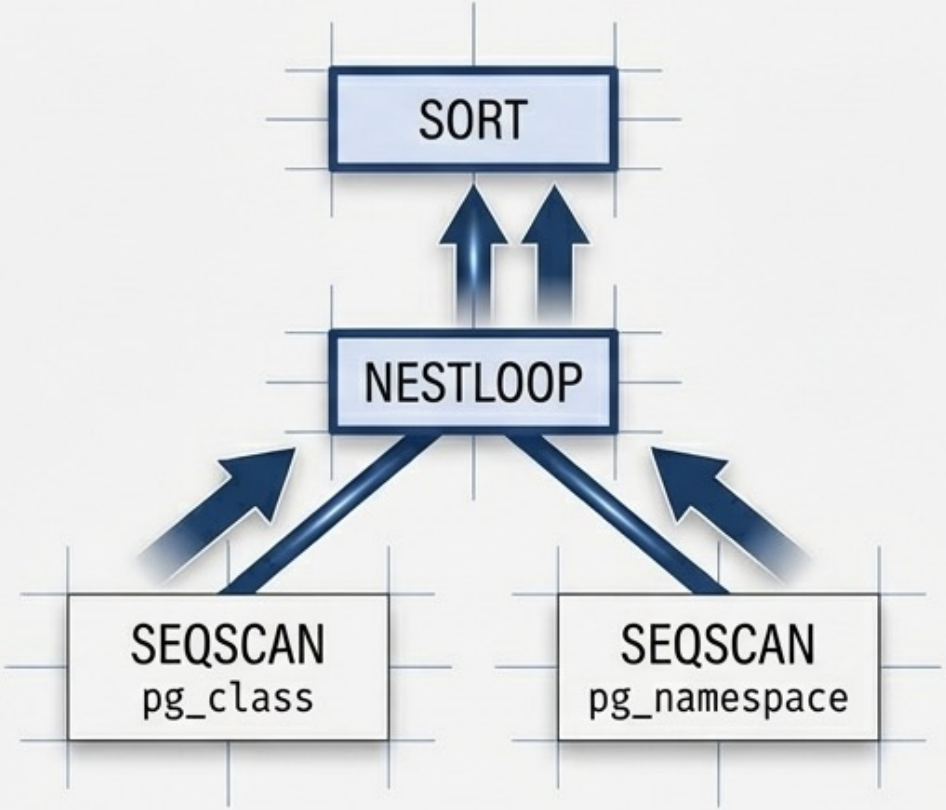
When the number of tables exceeds `geqo_threshold` (default: 12), Postgres switches to a genetic algorithm (GEQO) for a faster, 'good enough' plan rather than seeking a mathematically perfect one.

NotebookLM

Stage 4: Execution – Bringing the Plan to Life

The executor takes the chosen plan tree and executes it. The process works like a 'pull' pipeline:

1. The top-level node (in our case, SORT) requests a row from its child.
2. Its child (NESTLOOP) in turn requests rows from its children (SEQSCAN nodes).
3. The leaf nodes (SEQSCAN) fetch rows from the tables on disk.
4. Rows flow up the tree, being joined, filtered, and sorted as they go, until they reach the top and are sent to the client.



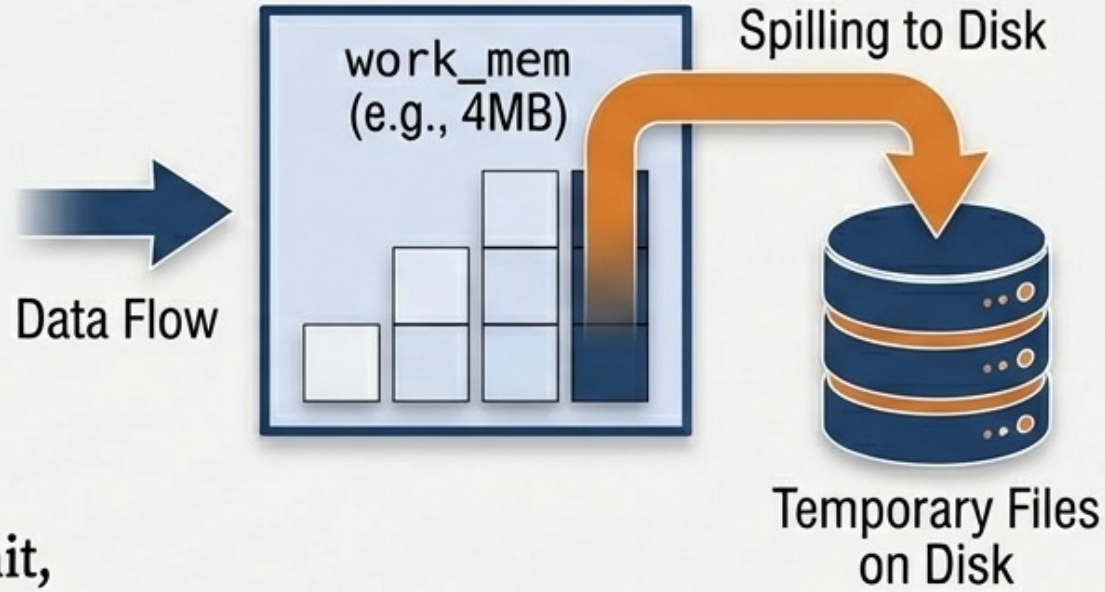
This node-by-node execution model is highly efficient. If a LIMIT clause is present, execution can stop as soon as enough rows have been produced, without processing the entire dataset.

NotebookLM

The Executor's Resources: Memory and Portals

During execution, the state of the query is held in an object called a **portal.**
held in an object called a **portal.**

Some plan nodes, like `Seq Scan` or `Index Scan`, stream rows without needing to store them. Others, like `SORT`, `HASH JOIN`, or certain aggregations, must materialize the entire dataset.



These 'blocking' operations are allocated memory up to the `work_mem` setting (default: 4MB). If the data exceeds this limit, the excess is 'spilled' to temporary files on disk, which can significantly slow down the query.

A single complex query can have multiple blocking nodes, each potentially allocating its own `work_mem` chunk.

NotebookLM

The Smarter Path: The Extended Query Protocol

The journey we've followed is the **Simple Query Protocol.** Every time a query is sent, it goes through all four stages: Parse, Rewrite, Plan, and Execute. This is inefficient for applications that run the same query structure repeatedly with different values.

The **Extended Query Protocol** offers a solution by breaking the process into discrete steps, allowing the expensive initial stages to be reused. This is the mechanism behind **prepared statements.**

Simple Protocol	Extended Protocol
Single 'Query' message	`Parse` -> `Bind` -> `Execute` messages
All 4 stages run every time	Parse & Rewrite once; Plan & Execute many times
Ideal for ad-hoc queries	Ideal for repetitive application queries

NotebookLM

Reusing the Work: Prepared Statements in Action

Preparing a statement allows the database to parse, analyze, and rewrite a query just once, caching the result.

```
PREPARE plane(text) AS
  SELECT * FROM aircrafts WHERE aircraft_code = $1;
```

When you `EXECUTE` the statement, you simply “bind” new parameter values.

```
EXECUTE plane('733');
EXECUTE plane('763');
```



****Crucial Security Benefit****

Parameter binding is not simple string concatenation. The parameter value is sent separately and can never alter the query's parsed structure. This makes prepared statements the gold standard for preventing SQL injection vulnerabilities.

NotebookLM

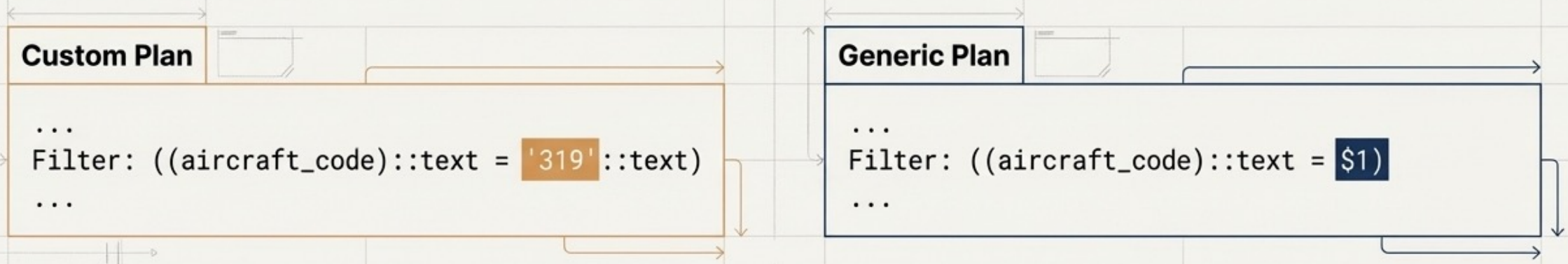
The Final Trade-Off: Custom vs. Generic Plans

When executing a prepared statement, the planner faces a choice:

- **Custom Plan:** Generate a new plan based on the *specific parameter values* provided. This is optimal when different values lead to vastly different data distributions (e.g., `total_amount > 1000000` vs. `total_amount > 100`).
- **Generic Plan:** Create a single plan that works for *any* parameter value (using `$1` as a placeholder). This avoids the overhead of replanning but may not be optimal for all values.

The Heuristic

By default (`plan_cache_mode = 'auto'`), PostgreSQL **creates custom** plans for the first 5 executions. On the 6th execution, it compares the average cost of the custom plans to the estimated cost of a generic plan. If the generic plan is cheaper (including the savings from not replanning), it switches to the generic plan for all future executions in that session.



NotebookLM

The Journey's End: From Theory to Practice

We've followed a query's transformation from a simple string into a complex, optimized execution plan.



Understanding this journey empowers you to:

- ✔ **Read `EXPLAIN` plans effectively**, connecting the output to the underlying physical operations.
- ✔ **Recognize the critical role of statistics**, ensuring `ANALYZE` is running properly to feed the planner good information.
- ✔ **Structure complex queries with joins** to guide the planner toward better paths.
- ✔ **Use prepared statements** in your applications to improve performance and security.

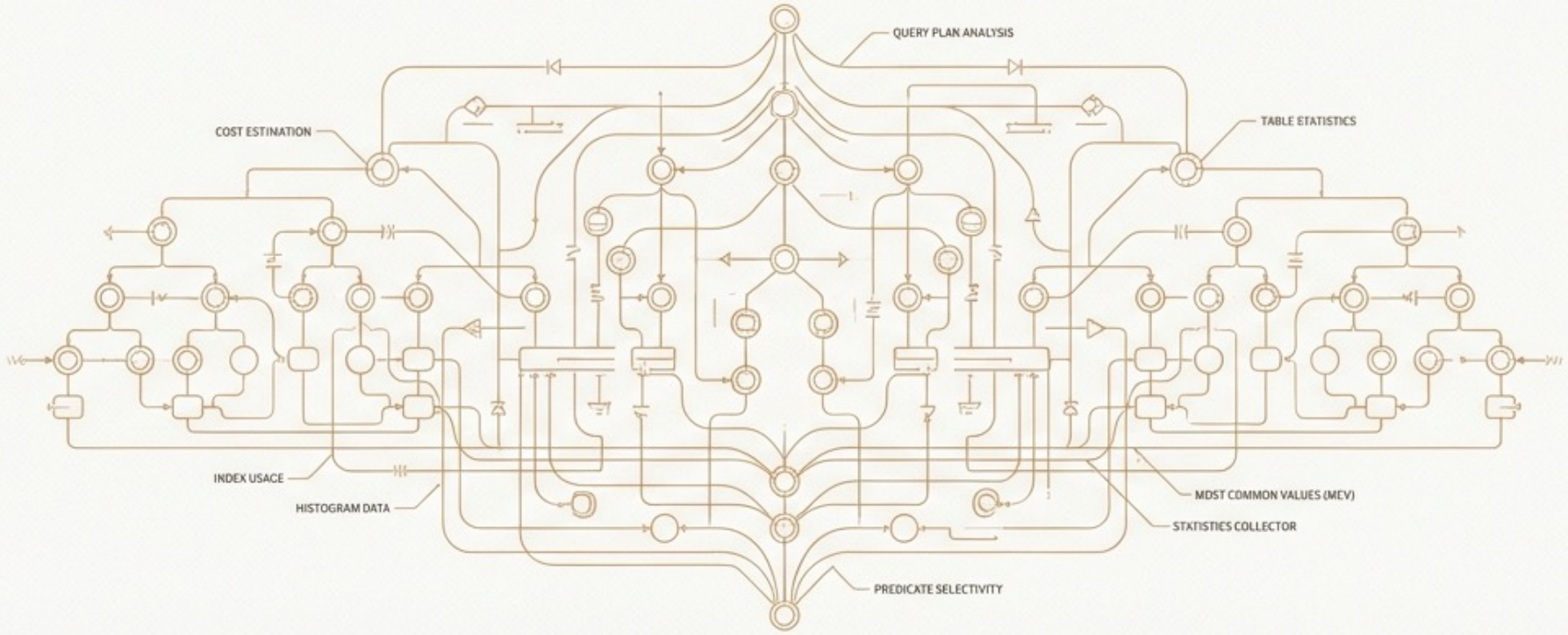
By knowing what happens 'under the hood,' you can move from simply writing SQL to engineering high-performance database interactions.

NotebookLM

17. Statistics

Deconstructing the Planner's Mind: A Deep Dive into PostgreSQL Statistics

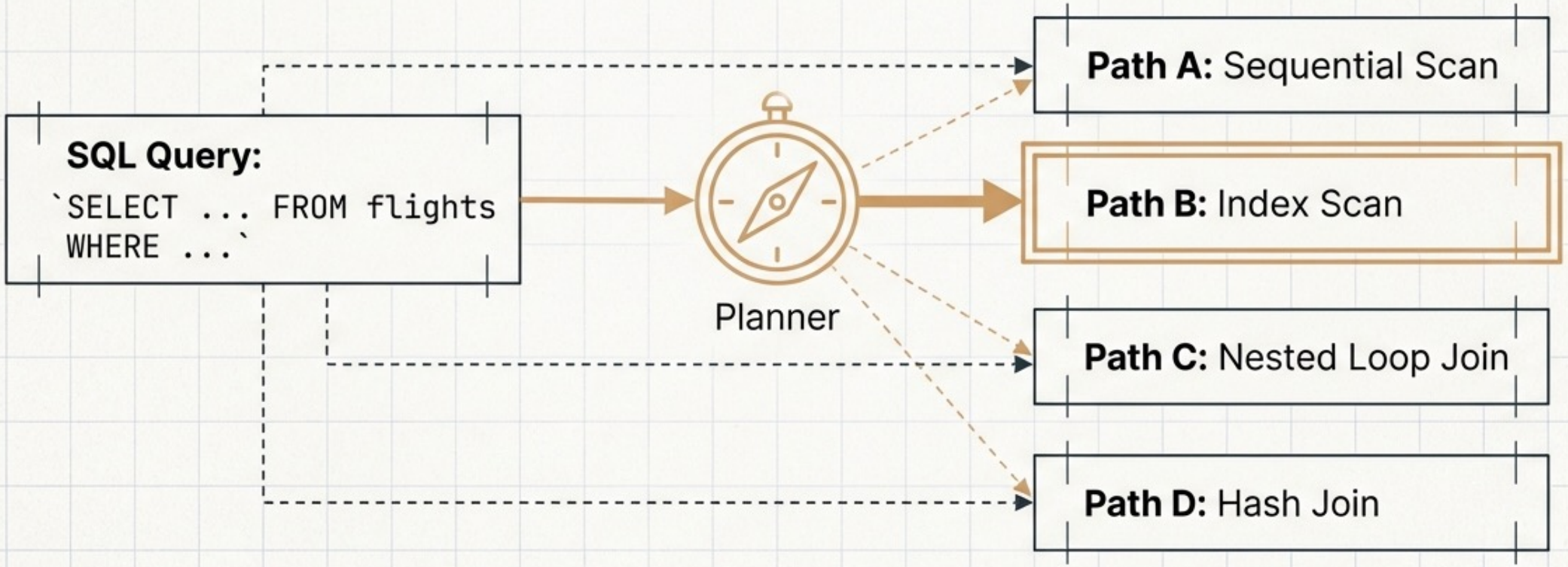
How the world's most advanced open-source database predicts the future of your queries.



NotebookLM

The Billion-Path Problem

A single SQL query can be executed in countless ways. The planner's job is to find the *cheapest* path without running them all.
Its currency is **Cost Estimation**. Its foundation is **Statistics**.



NotebookLM

Starting Macro: How Big is the Playground?

Problem: Before analyzing any conditions, the planner must know the scale of the table.

Solution: The planner consults basic stats stored in the `pg_class` catalog: `reltuples` (row count) and `relpages` (size in disk pages).

Proof:

```
SELECT reltuples, relpages
FROM pg_class
WHERE relname = 'flights';

-- reltuples | relpages
-------------+----------
--    214867 |      2624
```

```
EXPLAIN SELECT * FROM flights;

--                          QUERY PLAN
-- ------------------------------------------------------------
-- Seq Scan on flights (cost=0.00..4772.67 rows=214867 width=63)
```

NotebookLM

The Planner's Reality Check for Stale Stats

Problem: What happens if the table size doubles *after* the last `ANALYZE`? The `reltuples` statistic is now completely wrong.

Solution: The planner compares `relpages` (the size at the last `ANALYZE`) with the *actual* file size on disk. If there's a mismatch, it adjusts its row estimate accordingly.

Proof:

1. Initial State:

`flights_copy` table stats are stale:
`reltuples = 214867`,
`relpages = 2624`.

2. Action:

Rows are doubled, but no `ANALYZE` is run:
`INSERT INTO flights_copy SELECT * FROM flights;`
(Actual rows: 429,734).

3. Result (The `EXPLAIN` plan):

```
EXPLAIN SELECT * FROM flights_copy;
--                            QUERY PLAN
-- ------------------------------------------------------------------
-- Seq Scan on flights_copy (cost=0.00..9545.34 rows=429734 width=63)
```

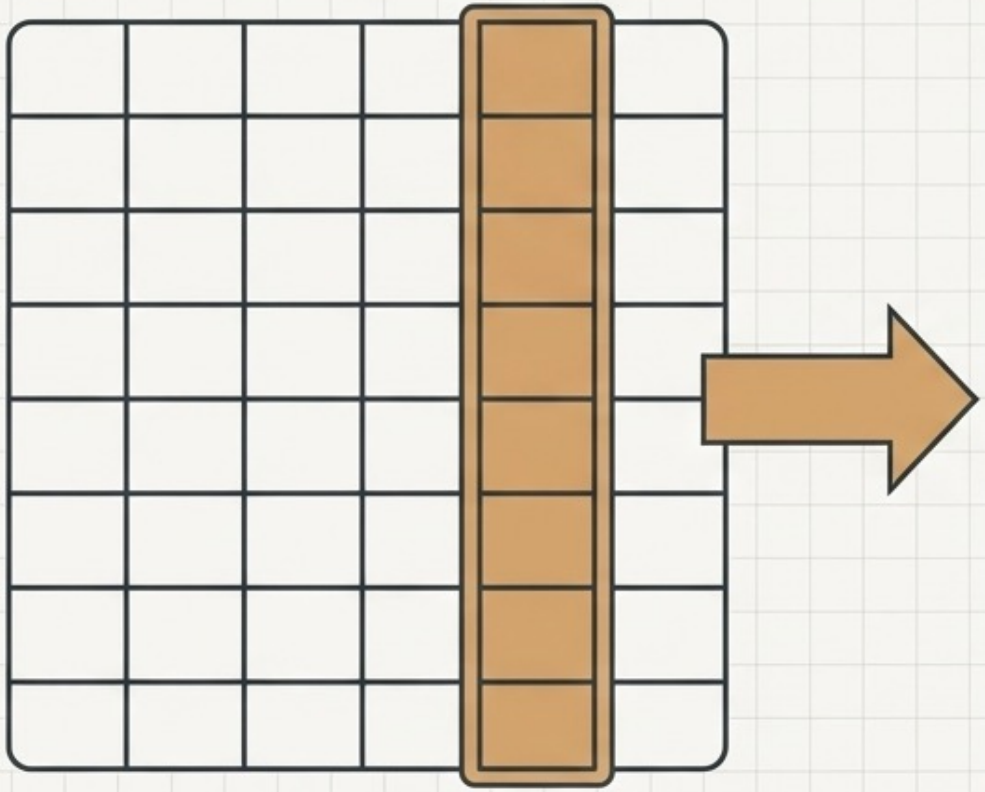
The Planner's Secret Adjustment:

```
adjusted_rows = reltuples * (actual_file_size_in_pages / relpages)
429734 = 214867 * (5248 / 2624)
```

NotebookLM

The Anatomy of a Column

Table size is not enough for filtering. For WHERE clauses, the planner must understand the data distribution within each column. This detailed information lives in the pg_stats view.



Column Statistical Profile (pg_stats)

- `null_frac` : ——— (The void)
- `n_distinct` : ——— (Variety)
- `most_common_vals` : — (The popular crowd)
- `most_common_freqs` : - (Popularity score)
- `histogram_bounds` : — (The long tail)
- `correlation` : ——— (Physical vs. logical order)
- `avg_width` : ——— (Average size)

NotebookLM

Challenge #1: Handling the Unknown with `null_frac`

Problem: How many flights have not yet departed? The planner needs to estimate the result of `WHERE actual_departure IS NULL`.

Solution: The `null_frac` statistic, which stores the simple percentage of rows in a column that are NULL.

Proof:

```
EXPLAIN SELECT *
FROM flights
WHERE actual_departure IS NULL;

--                QUERY PLAN
-- ------------------------------------------------------
-- Seq Scan on flights (cost=0.00..4772.67 rows=16702 width=63)
--   Filter: (actual_departure IS NULL)
```

The Calculation:

Estimated Rows = **reltuples * null_frac**

16702 = 214867 * 0.07773...

(**from** `pg_stats` **where** `attname` = 'actual_departure')

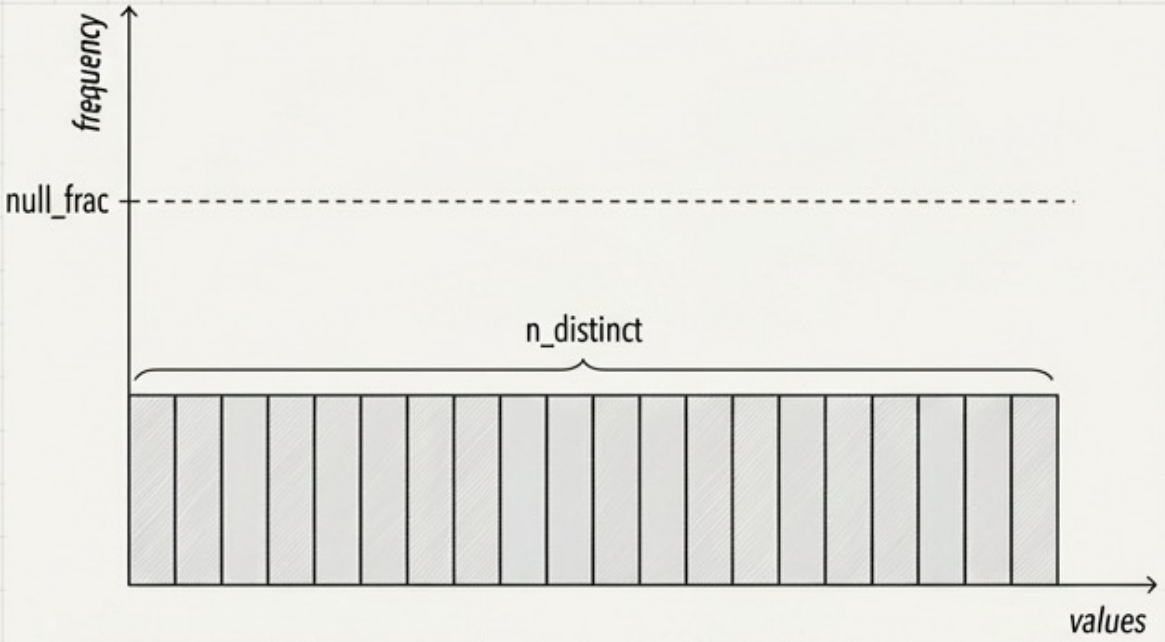
Actual count: 16,348

NotebookLM

Challenge #2: The Ideal World of `n_distinct`

Problem: How many flights departed from a *specific*, but unknown, airport? (`WHERE departure_airport = ?`). Without knowing the value, what is the best guess?

Solution: The planner's default assumption is that any value is equally likely. It uses `n_distinct` (the number of distinct values) to calculate selectivity as `1 / n_distinct`.



The Planner's Ideal World: Uniform Distribution.

Proof:

```
EXPLAIN SELECT *
FROM flights
WHERE departure_airport = (...); -- (value unknown to planner)

--                          QUERY PLAN
-- ---------------------------------------------------------------
-- Seq Scan on flights (cost=30.56..5340.40 rows=2066 width=63)
--   Filter: (departure_airport = $0)
```

The Calculation:

Estimated Rows = **reltuples / n_distinct**
2066 = 214867 / 104

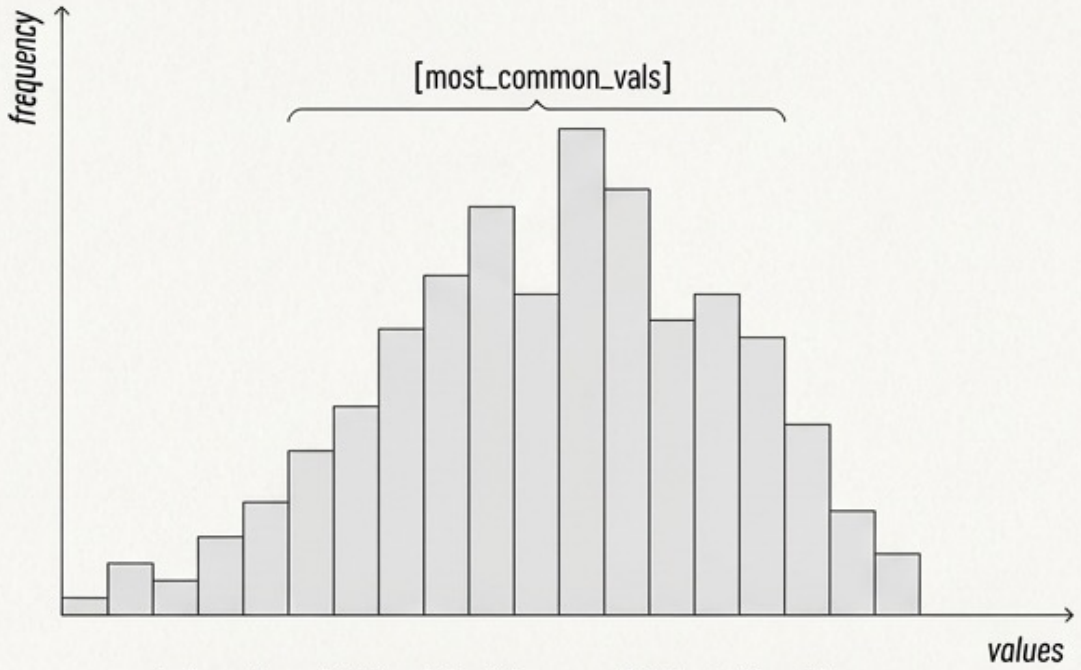
(n_distinct for departure_airport is 104)

NotebookLM

Challenge #3: Mastering Skew with Most Common Values

Problem: The uniform assumption is often wrong. Some airports are massive hubs (max flights: 20,875); others are tiny. The average (2,066) is misleading for common values.

Solution: The planner keeps a 'cheat sheet' of the most popular values and their exact frequencies, stored in `most_common_vals` and `most_common_freqs`.



The Real World: Skewed Distribution.

Proof:

```
EXPLAIN SELECT * FROM flights
WHERE aircraft_code = '733';

--                            QUERY PLAN
-- ---------------------------------------------------------------
-- Seq Scan on flights (cost=0.00..5309.84 rows=8093 width=63)
--   Filter: (aircraft_code = '733'::bpchar)
```

The Calculation:

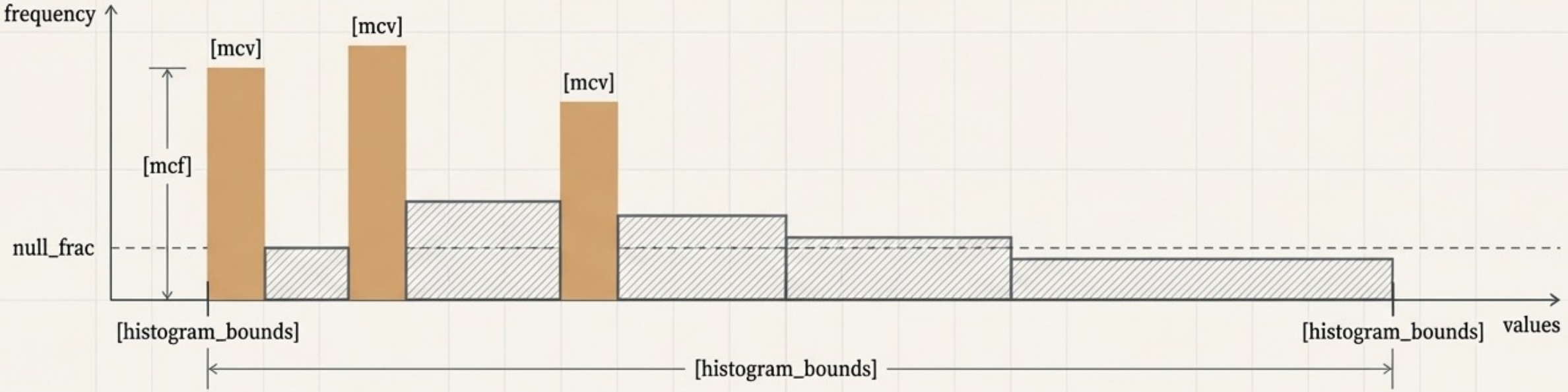
Estimated Rows = reltuples * most_common_freqs[index_of_'733']
8093 = 214867 * 0.03766...
(Actual count: 8,263)

NotebookLM

Challenge #4: Handling the Long Tail with Histograms

Problem: What if there are too many distinct values to fit in the MCV list, like timestamps or seat numbers? The planner can't store the frequency of every single value.

Solution: Histograms. The planner "pulls out" the MCVs. For the rest of the data, it groups values into buckets of roughly equal population, storing only the boundary values.



Proof:

```
EXPLAIN SELECT * FROM boarding_passes WHERE seat_no > '30B';

--                              QUERY PLAN
-- ---------------------------------------------------------------------
-- Seq Scan on boarding_passes (cost=0.00..157350.10 rows=2983242 ...)
--   Filter: ((seat_no)::text > '30B'::text)
```

The Combined Logic:

Selectivity = (Fraction of MCVs in range) + (Fraction of histogram buckets in range)

Estimated Rows = Total Rows * (MCV_share + (1 - total_mcv_freq) * Histogram_share)
2,983,242 = 8,760,042 * (0.2122 + (1 - 0.6781) * (51 / 100))

NotebookLM

Thinking Beyond Columns: Statistics on Expressions

Problem: The planner is blind to the meaning of functions. A condition like `WHERE extract(month from...) = 1` is a black box, forcing a pessimistic default guess of 0.5% selectivity.

Solution: `CREATE STATISTICS`. This command explicitly tells Postgres to analyze and collect statistics on the *result* of an expression, just like a regular column.

Proof:

BEFORE `CREATE STATISTICS`

```
EXPLAIN SELECT * FROM flights
WHERE extract(month from ...) = 1;

-- Seq Scan on flights (... rows=1074...)
```

Incorrect Estimate

THE SOLUTION

```
CREATE STATISTICS flights_expr
ON (extract(...)) FROM flights;

ANALYZE flights;
```

AFTER `CREATE STATISTICS`

```
EXPLAIN SELECT * FROM flights
WHERE extract(month from ...) = 1;

-- Seq Scan on flights (... rows=16667...)
```

Correct Estimate

NotebookLM

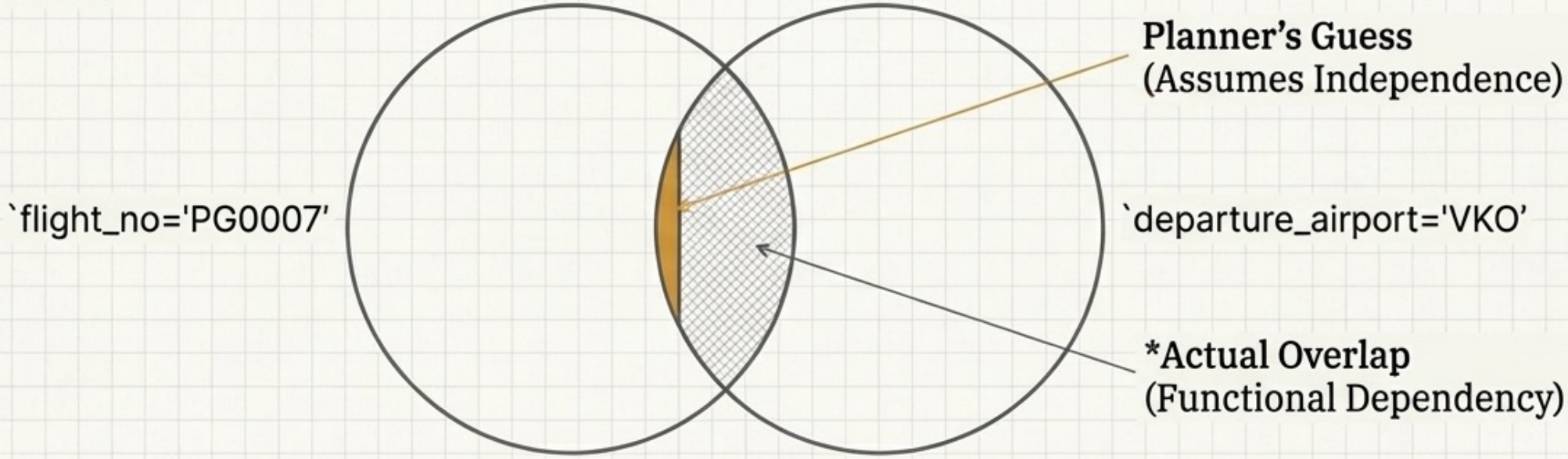
When Columns Don't Act Alone: Multivariate Statistics

Until now, we've treated columns as independent. But what if they are strongly related? The planner's default behavior—multiplying the selectivities of individual WHERE clauses—will fail dramatically. Sel(A AND B) = Sel(A) * Sel(B) is often wrong.

Problem Scenario

```
WHERE flight_no = 'PG0007' AND departure_airport = 'VKO'
```

A flight number determines the departure airport. They are not independent filters.



NotebookLM

Case Study 1: Teaching the Planner about Dependencies

Solution: Create extended statistics to track (dependencies) between columns. This tells the planner how strongly one column's value determines another's.

Proof (Before/After `CREATE STATISTICS`)

BEFORE (Assumes Independence)

```
EXPLAIN SELECT * FROM flights
  WHERE flight_no='PG0007'
    AND departure_airport='VKO';

-- Bitmap Heap Scan on flights (... rows=15 ...)
```

Incorrect Estimate

**The Command:

```
CREATE STATISTICS flights_dep(dependencies)
ON flight_no, departure_airport FROM flights;
```

Resulting stat: `{"2 => 5": 1.000000}`
(Column 5 is fully determined by Column 2)

AFTER (Understands Dependency)

```
EXPLAIN SELECT * FROM flights
  WHERE flight_no='PG0007'
    AND departure_airport='VKO';

-- Bitmap Heap Scan on flights (... rows=277 ...)
```

Correct Estimate

NotebookLM

Case Study 2: Finding Common Pairs and Combinations

Problem 1: `GROUP BY` on multiple columns

How many unique (`departure_airport, arrival_airport)` pairs exist? The planner's guess (`n_distinct(dep) * n_distinct(arr)`) is often a huge overestimate.

Proof (`ndistinct`):

Before:

Incorrect Estimate

```
HashAggregate (... rows=10816 ...)
```

After:

Correct Estimate

```
HashAggregate (... rows=618 ...) (Actual: 618)
```

Problem 2: Common Combinations

Some value pairs are far more common. `departure_airport = 'SVO'` and `aircraft_code = '733'` is a frequent combination that individual MCVs would not capture.

Proof (`mcv`):

Before:

Incorrect Estimate

```
Seq Scan (... rows=736 ...)
```

After:

Correct Estimate

```
Seq Scan (... rows=1927 ...) (Actual: 2037)
```

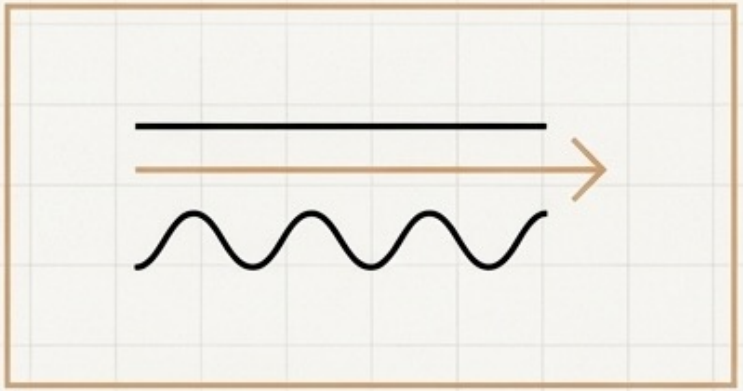
`CREATE STATISTICS` can be used to track `ndistinct` and `mcv` on groups of columns, dramatically improving estimates for multi-column `GROUP BY` and `WHERE` clauses.

NotebookLM

Architect's Blueprint

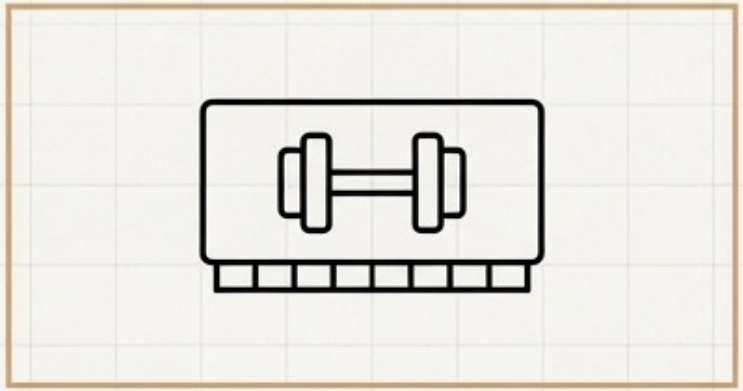
The Unsung Heroes of Costing

Cardinality (row count) is critical, but it's not the only thing the planner cares about. Other statistics are vital for estimating the *cost* of operations.



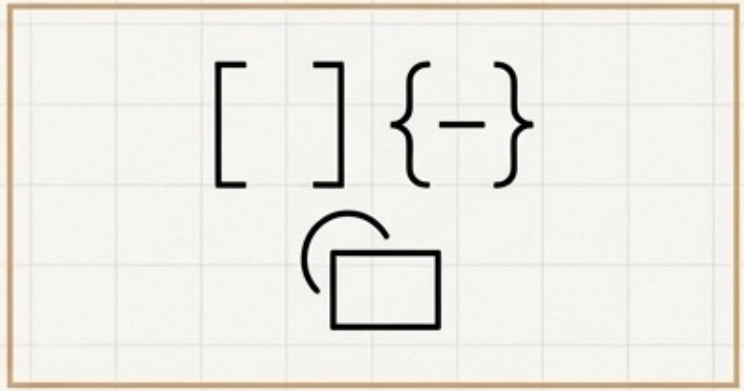
`correlation`

Measures the relationship between the physical order of data on disk and its logical (value) order. **Crucial for estimating the cost of an Index Scan.** High correlation means fewer random disk I/Os.



`avg_width`

The average size (in bytes) of data in a column. **Essential for estimating memory needed for sorts, hash tables, and joins.**



Non-Scalar Stats

Postgres also collects specialized statistics for complex types like arrays (`most_common_elems`), ranges (`range_length_histogram`), and tsvectors.

NotebookLM

You Are the Planner's Guide

The PostgreSQL planner is incredibly sophisticated, but it's not magic. It's a cost-based optimizer that relies entirely on the quality of the statistics you provide. Understanding this toolkit is the key to mastering performance.

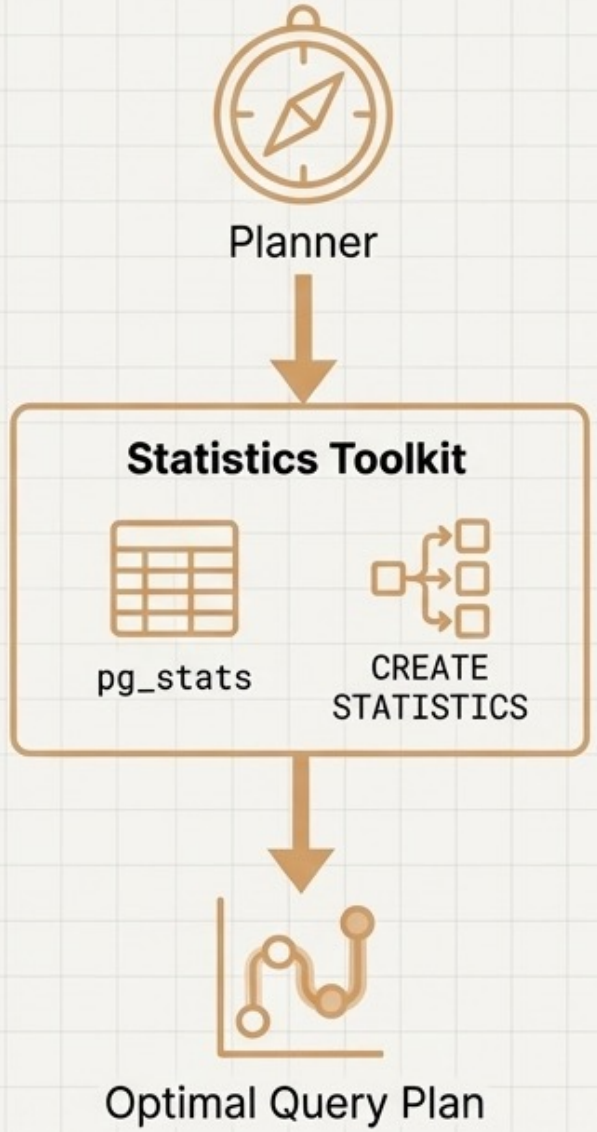
What to Do Next:

1. **Trust but Verify** (`EXPLAIN ANALYZE`): Look for large discrepancies between `rows` (`estimated`) and `actual rows`. This is your primary clue that statistics are misleading the planner.

2. `ANALYZE` **is Your Friend:** Ensure autovacuum is running effectively to keep stats reasonably up-to-date. `ANALYZE` manually after major data loads.

3. **Deploy the Advanced Toolkit:** For correlated columns (`dependencies`), functions in `WHERE` clauses (`expressions`), or multi-column filters (`mcv`), don't hesitate to `CREATE STATISTICS`.

4. **Tune the Target:** For critical columns with complex distributions, selectively increase the detail by using `ALTER TABLE ... SET STATISTICS ...`. This increases the sample size and the size of the MCV/histogram lists.

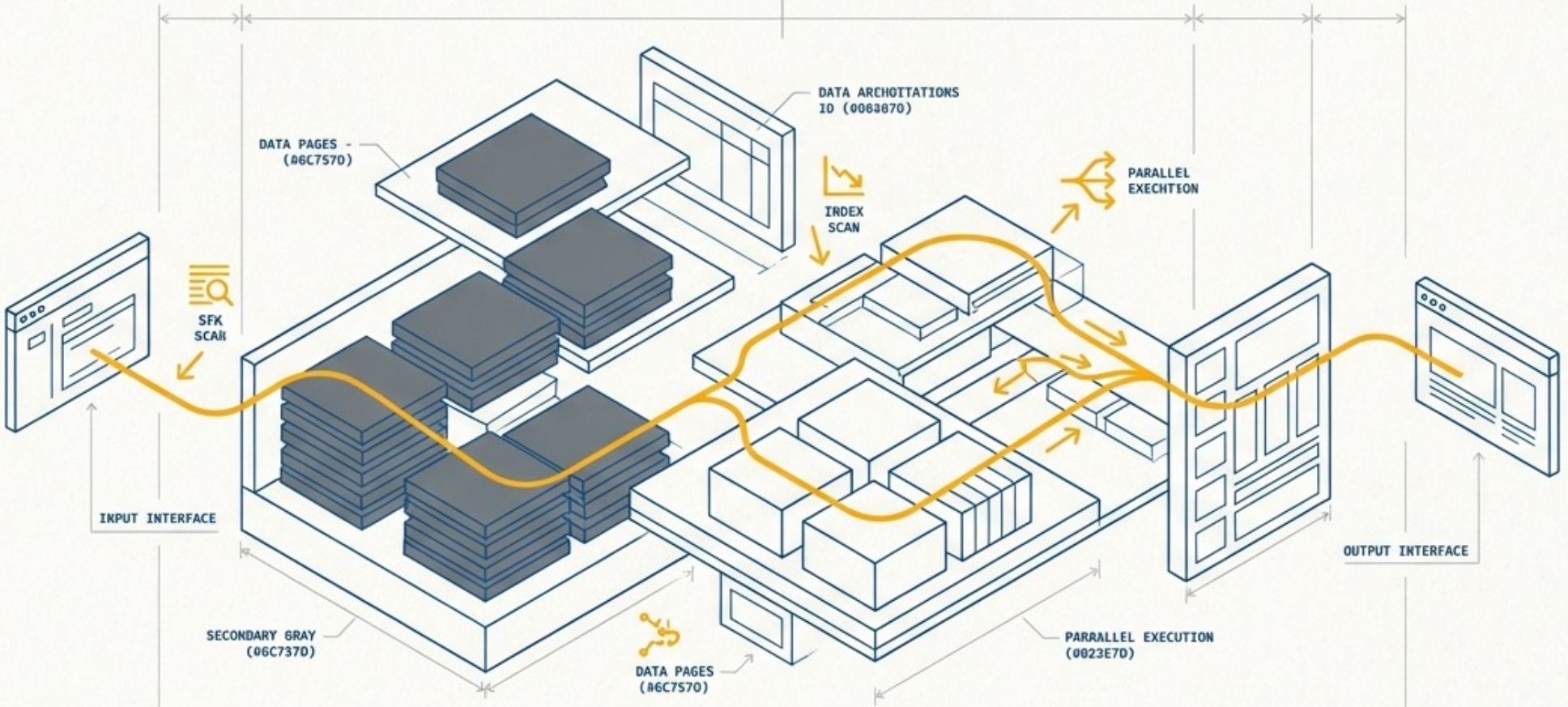


NotebookLM

18. Table Access Methods

PostgreSQL: Under the Hood of Table Access

From Sequential Scans to Parallel Execution

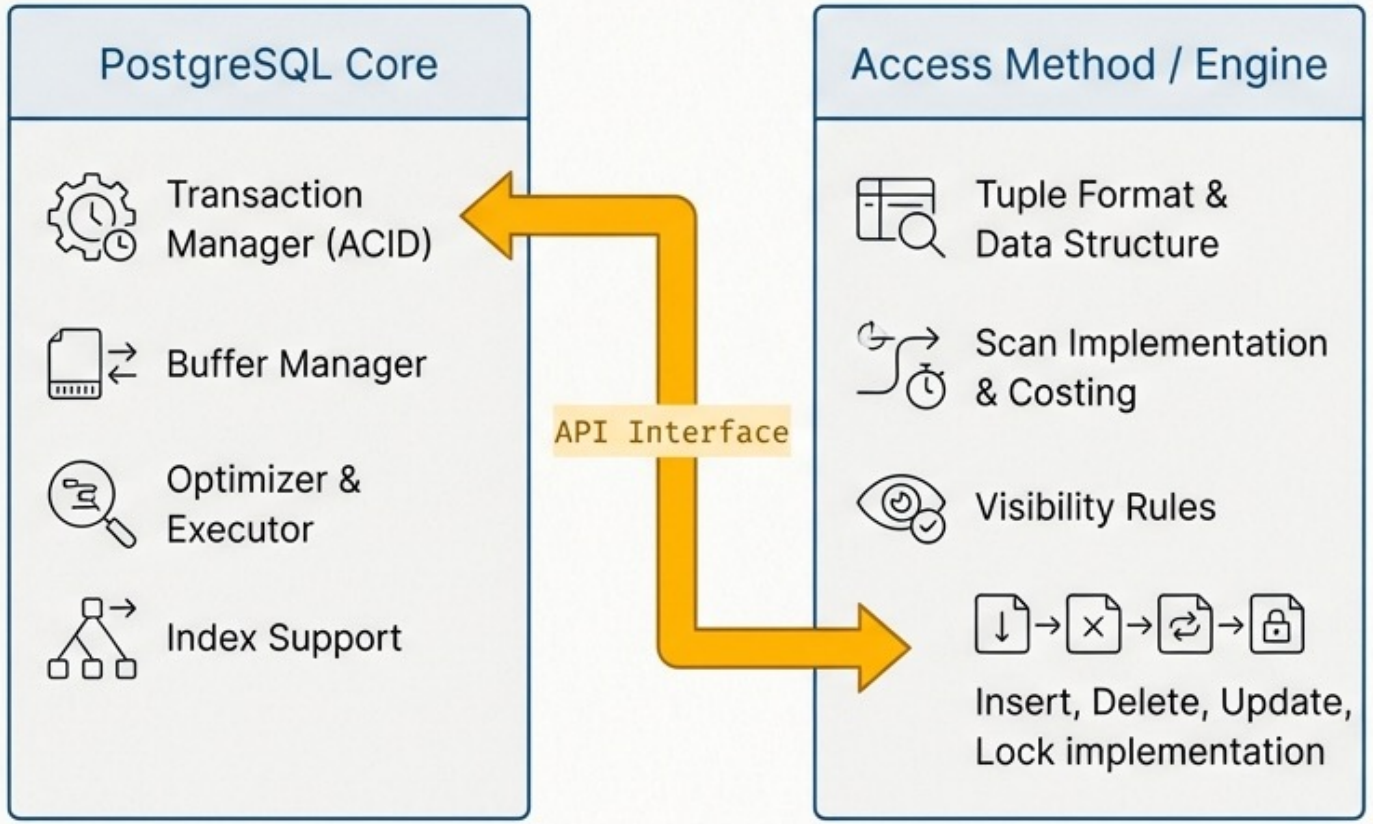


NotebookLM

PostgreSQL is built on a pluggable storige engine architecture.

PostgreSQL supports different "Table Access Methods," which are essentially pluggable storage engines. The engine is specified using the `USING` clause in `CREATE TABLE`. The default is `heap`.

This design creates a formal separation between the core database services and the storage layer's responsibilities.



NotebookLM

The tight coupling with ‘heap’ presents challenges, but new engines are emerging.

Challenge

Deep integration points, such as WAL logging, create a strong bond between the core and the default heap engine, making it difficult to create fully independent new engines.

On the Horizon



Zheap

Aims to solve table bloat. It uses in-place updates and moves old data to separate undo storage. Ideal for high-update OLTP workloads.



Zedstore

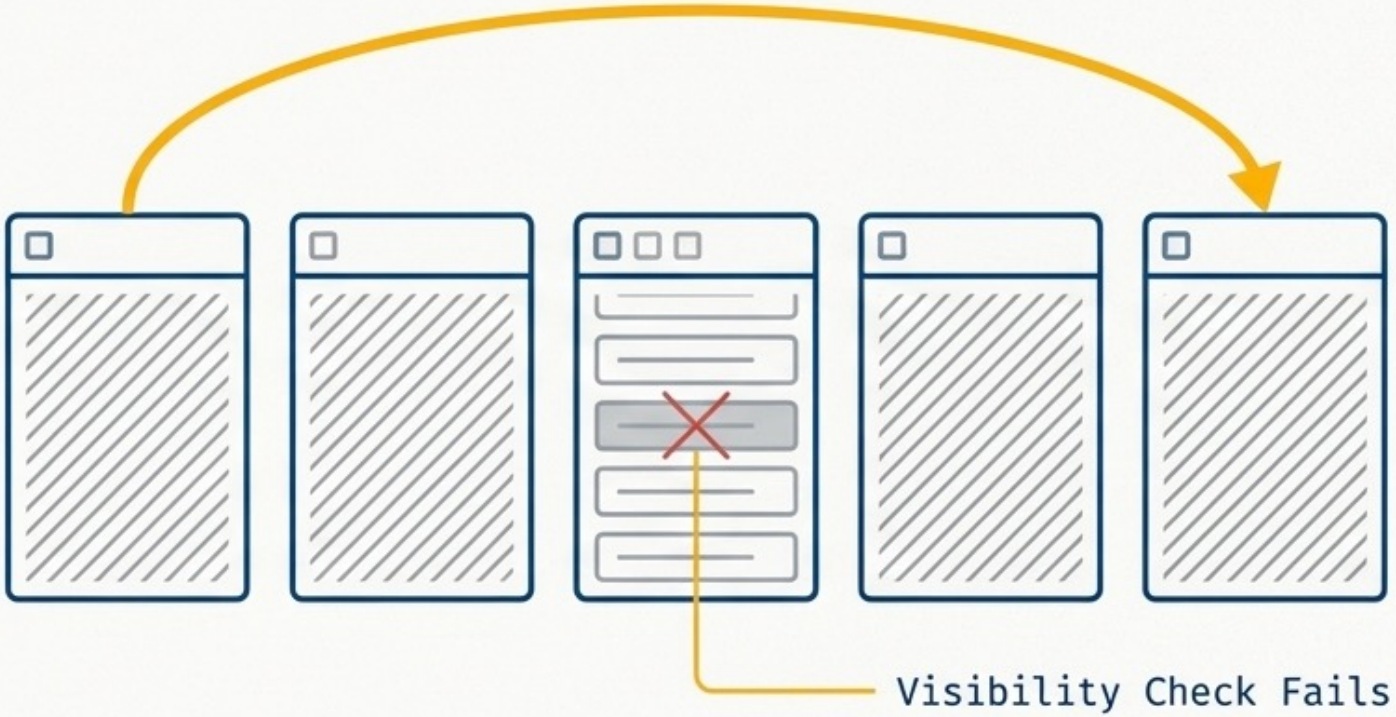
A columnar store designed for high-performance OLAP (Online Analytical Processing) queries.

NotebookLM

The Sequential Scan is the fundamental method for reading an entire table.

A sequential scan reads the table's main data file(s) from beginning to end. For each tuple on every page, it performs a visibility check against the current transaction's snapshot. Tuples that are not visible or do not match the query's filter conditions are discarded.

This is the most efficient method when selectivity is low (i.e., when a large portion of the table needs to be read).



NotebookLM

The optimizer's cost estimate for a Seq Scan is a sum of I/O and CPU work.

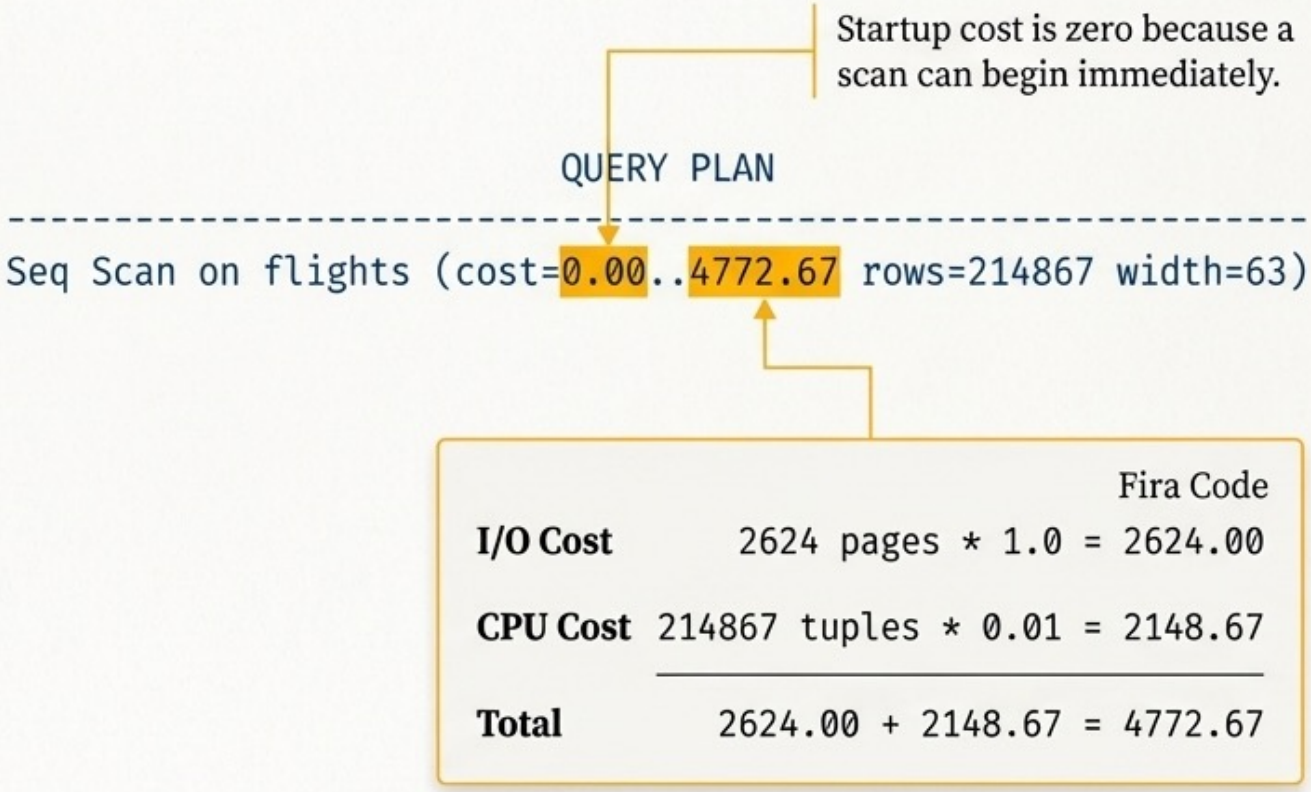
Total Cost = I/O Cost + CPU Cost

I/O Cost: `relpages * seq_page_cost`

- The `seq_page_cost` (1.0) is lower than `random_page_cost` (4.0) because the OS can prefetch data during sequential reads.

CPU Cost: `reltuples * cpu_tuple_cost`

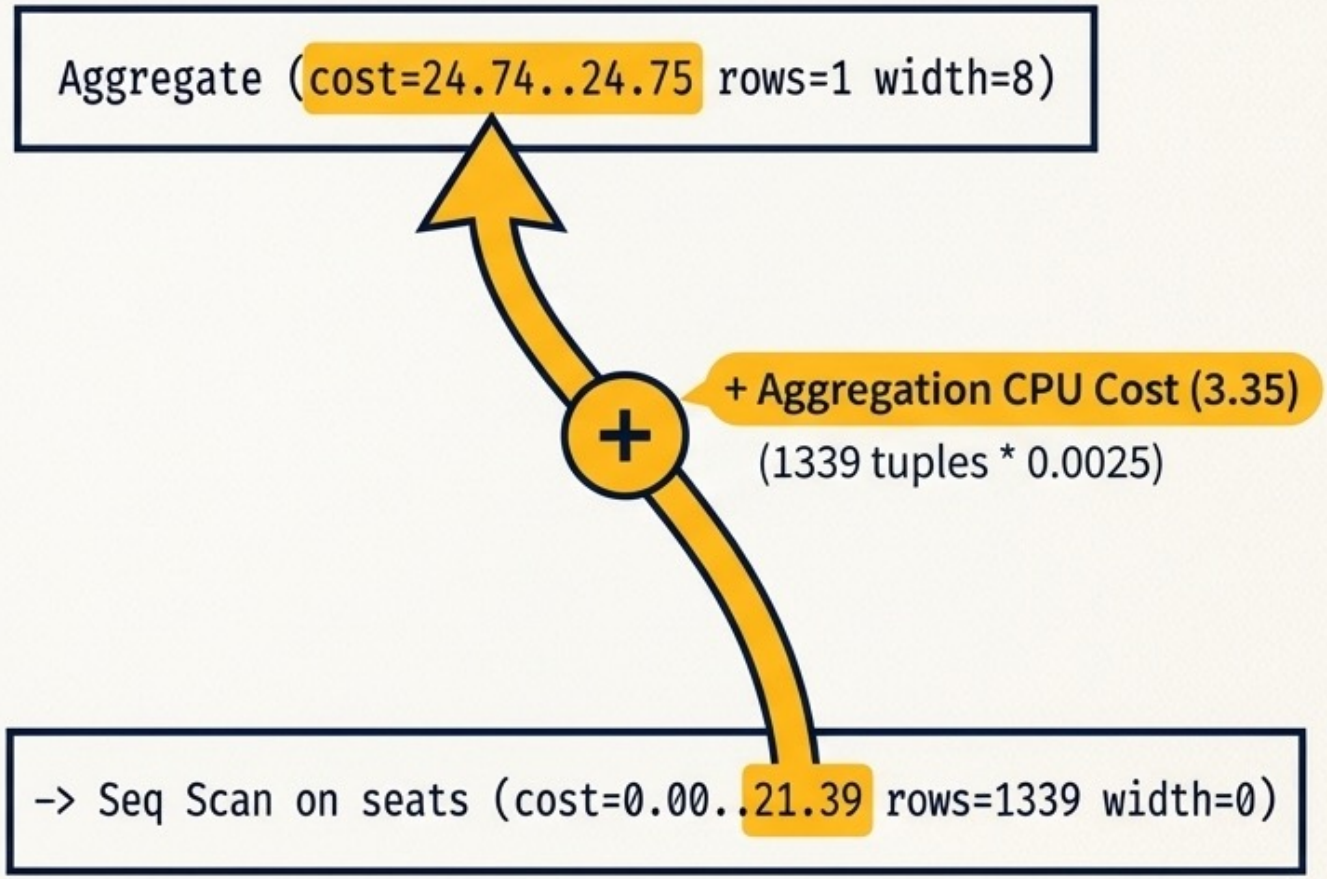
- This represents the work to process each tuple (default: 0.01).



NotebookLM

Costs flow up the plan tree, with each node adding its own operational cost.

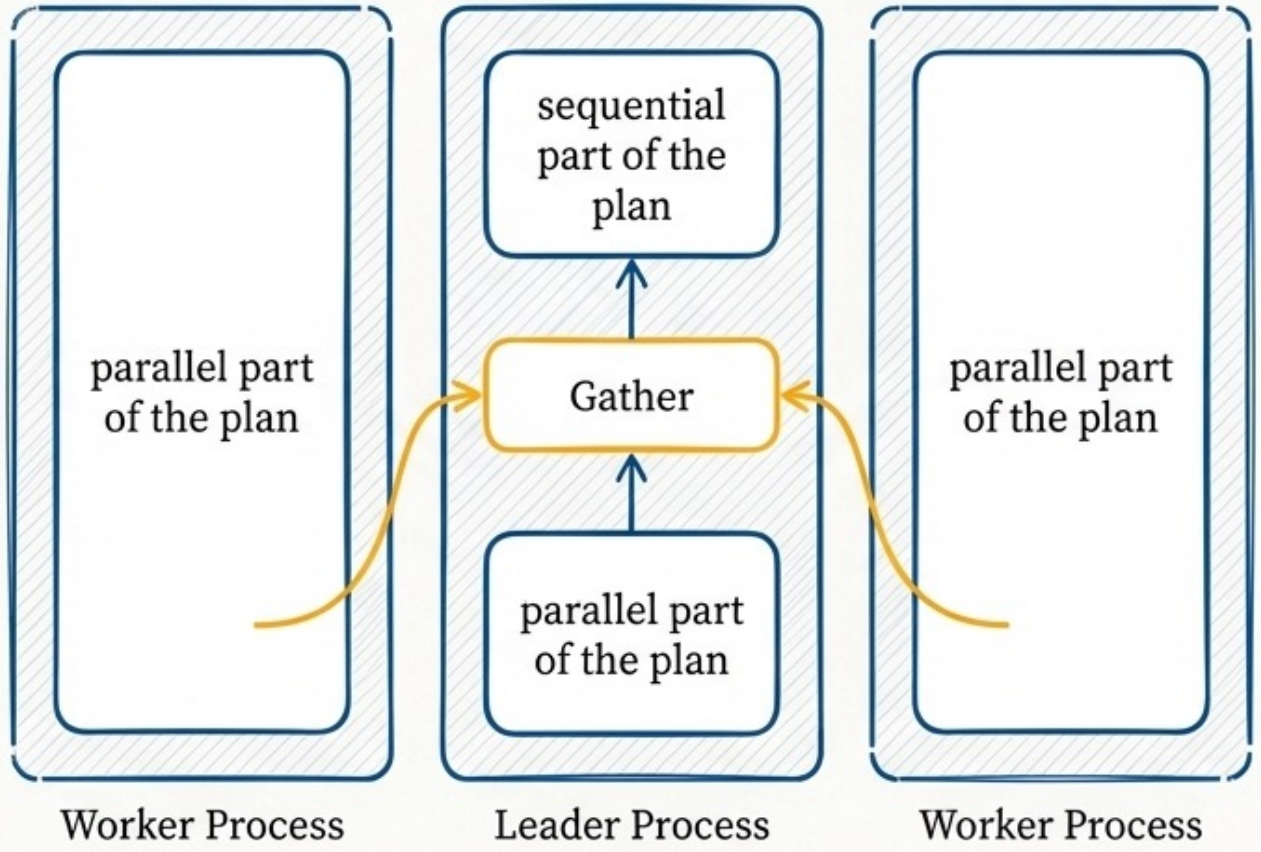
In a plan tree, parent nodes inherit costs from their children. An Aggregate node is a “blocking” operator; it cannot produce its result until it has consumed all input rows from the child Seq Scan. Therefore, the Startup Cost of the Aggregate must include the Total Cost of the Seq Scan, plus the CPU cost of the aggregation work (cpu_operator_cost).



NotebookLM

PostgreSQL can execute parts of a query in parallel using a Leader and Workers.

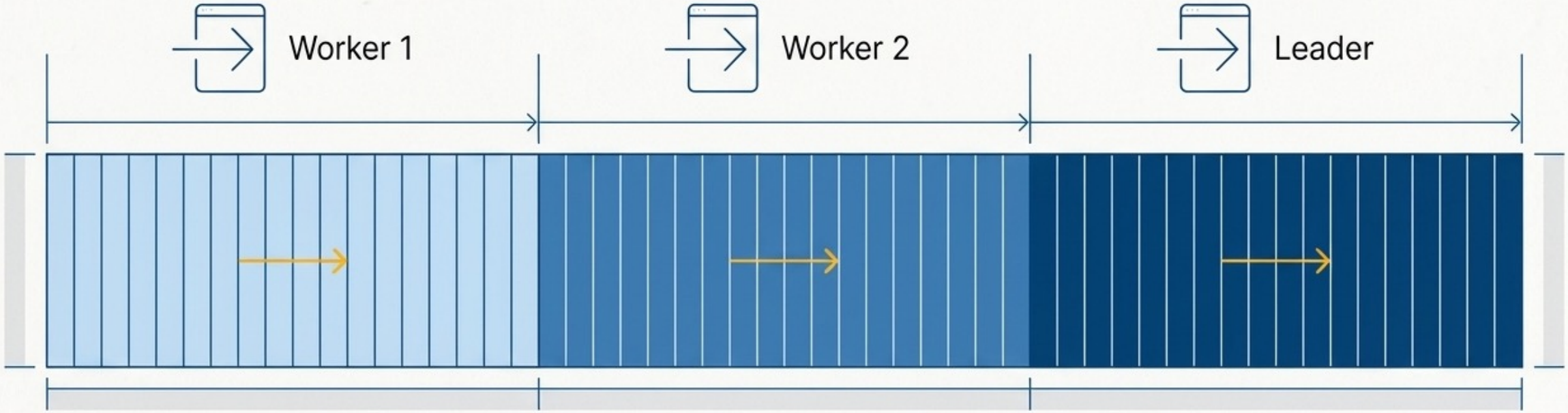
- A single leader process orchestrates the query.
- The leader spawns multiple background worker processes.
- Workers execute the parallel portion of the plan concurrently.
- The `Gather` node, running in the leader, collects results from all workers and passes them to the sequential part of the plan.



NotebookLM

A Parallel Seq Scan divides the work of a full table scan among multiple processes.

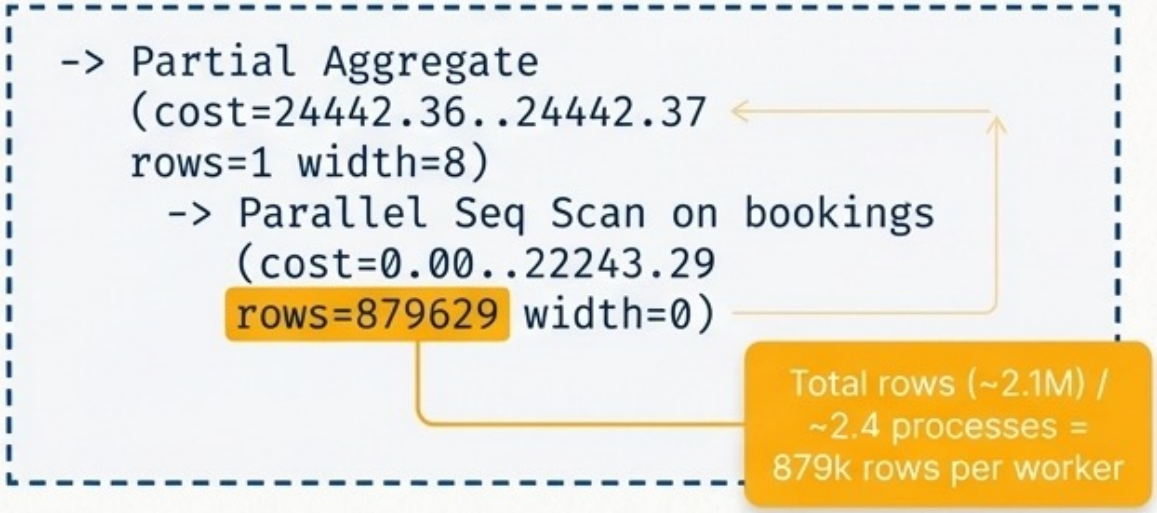
- The table's pages are logically divided and assigned to the leader and workers.
- Each process sequentially scans its own assigned portion of the pages.
- Shared memory is used to synchronize progress and prevent scanning the same page twice.
- **Key Trade-off**: While CPU work is parallelized, disk I/O can appear more random to the OS, potentially reducing prefetching. PostgreSQL mitigates this by assigning contiguous blocks of pages to each worker.



NotebookLM

The parallel portion of the plan is executed by each worker on a subset of the data.

- The plan nodes appearing below the `Gather` node constitute the parallel part.
- The cost for `Parallel Seq Scan` is calculated similarly to a regular `Seq Scan`, but on a smaller number of rows (`total_rows / num_processes`).
- `Partial Aggregate` is a crucial optimization: each worker aggregates its local results before sending a much smaller dataset to the leader.



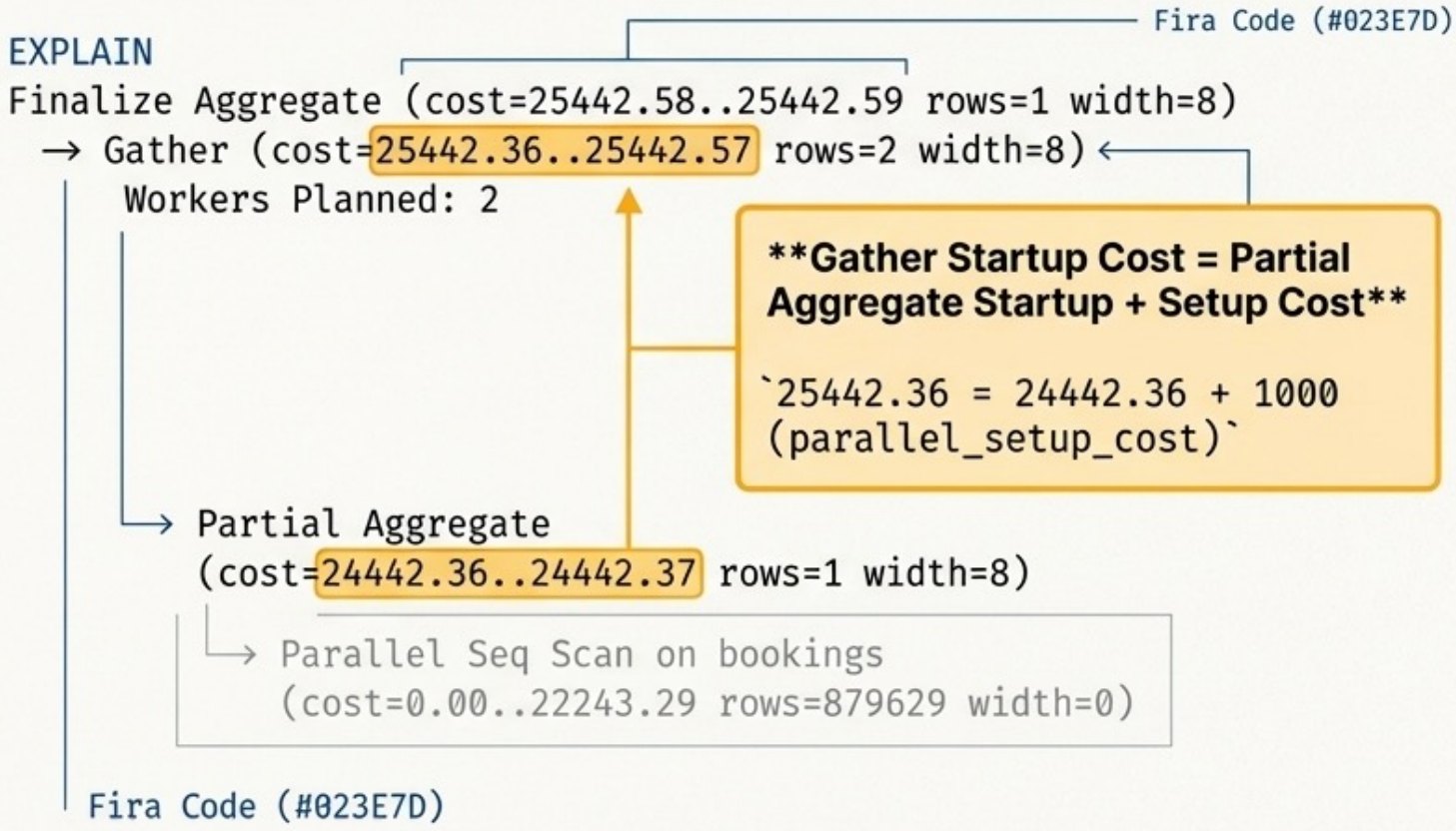
NotebookLM

The leader gathers results and finalizes the calculation, adding overhead costs.

The Gather node introduces two new costs specific to parallelism:

- `parallel_setup_cost` (default: 1000): The fixed cost of launching worker processes.
- `parallel_tuple_cost` (default: 0.1): The cost to transfer each tuple from a worker to the leader.

The `Finalize Aggregate` node performs the final aggregation on the partial results received from all workers.



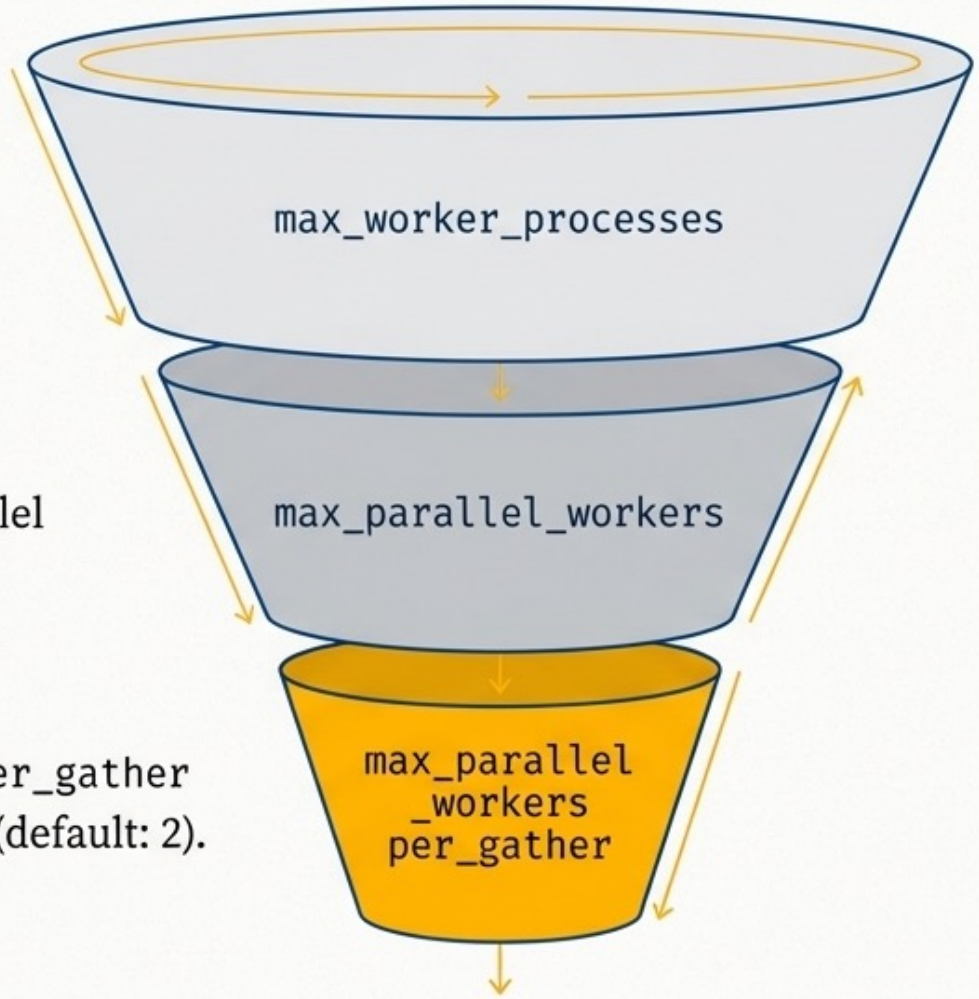
NotebookLM

The number of workers is governed by a hierarchy of configuration parameters.

System-Wide Limit: `max_worker_processes`
Total background workers available on the system (default: 8).

Parallelism Limit: `max_parallel_workers`
Maximum workers that can be used for parallel queries (default: 8).

Per-Query Limit: `max_parallel_workers_per_gather`
Maximum workers for a single `Gather` node (default: 2).



Optimal values depend on available CPU cores, workload type (OLAP vs. OLTP), and typical table sizes.

NotebookLM

The planner automatically determines the optimal number of workers for a given table.

A parallel plan is only considered if the table is larger than `min_parallel_table_scan_size` (default: 8MB). The number of planned workers is calculated logarithmically based on the table size, up to the configured limit.

Rule of Thumb: Postgres adds one worker for every 3x increase in table size, starting from the 8MB threshold.

$$1 + \left\lfloor \log_3 \left(\frac{\text{table size}}{\text{min_parallel_table_scan_size}} \right) \right\rfloor$$

Table Size →	Planned Workers
8MB →	1
24MB →	2
72MB →	3 (if limit allows)
216MB →	4 (if limit allows)

NotebookLM

Not all queries can be parallelized, even if they seem like good candidates.



Data Modification: `UPDATE`, `DELETE`, and `SELECT FOR UPDATE` cannot have parallel plans.



Pausable Queries: Cursors are non-parallel, which includes implicit cursors in PL/pgSQL `FOR` loops.



Unsafe Functions: Any query that calls a function marked `PARALLEL UNSAFE`. This is the default for user-defined functions.

Nested Parallelism: A query inside a function that is called from an *already* parallel query will not itself be parallelized.

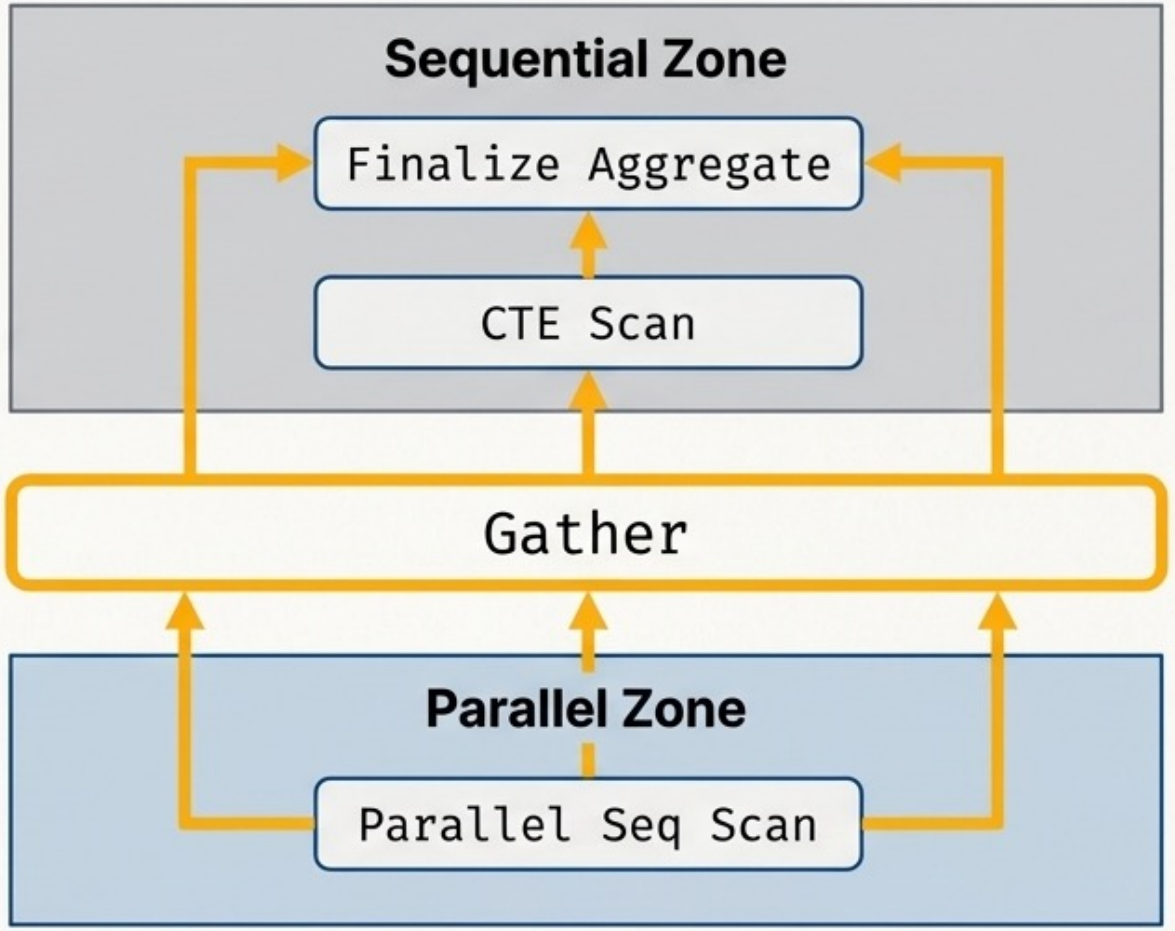
NotebookLM

Some operations are “parallel-restricted” and must run in the sequential part of a plan.

Certain plan nodes are not permitted to execute below a `Gather` node (i.e., within a worker process). This forces any part of the plan containing them to be sequential, creating a ‘sequential zone’ at the top of the plan.

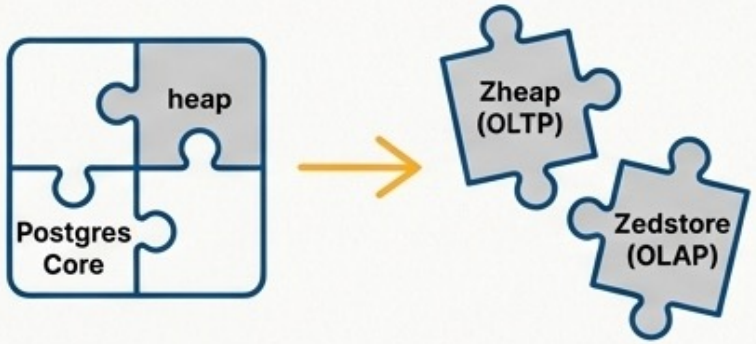
Common Examples:

- `CTE Scan`: Scanning the results of a materialized CTE.
- `SubPlan` / `InitPlan`: Correlated or independent subqueries.
- Scans on temporary tables.



NotebookLM

Architecting for Performance: Key Takeaways on Table Access



Pluggable Future

PostgreSQL's `heap` engine is deeply integrated, but a future with specialized engines like Zheap (OLTP) and Zedstore (OLAP) is on the horizon.

```
EXPLAIN
Seq Scan on table_name (cost=0.00..1234.56 rows=100000 width=32)
```

Cost = I/O + CPU

relpages reltuples

Sequential Scan Baseline

Sequential scans are the efficient foundation. Their cost is a predictable formula of I/O (`relpages`) and CPU (`reltuples`).



Parallel Scan Purpose

Parallel scans excel on large tables by parallelizing CPU-intensive work (like aggregations), which is often the primary benefit over I/O.



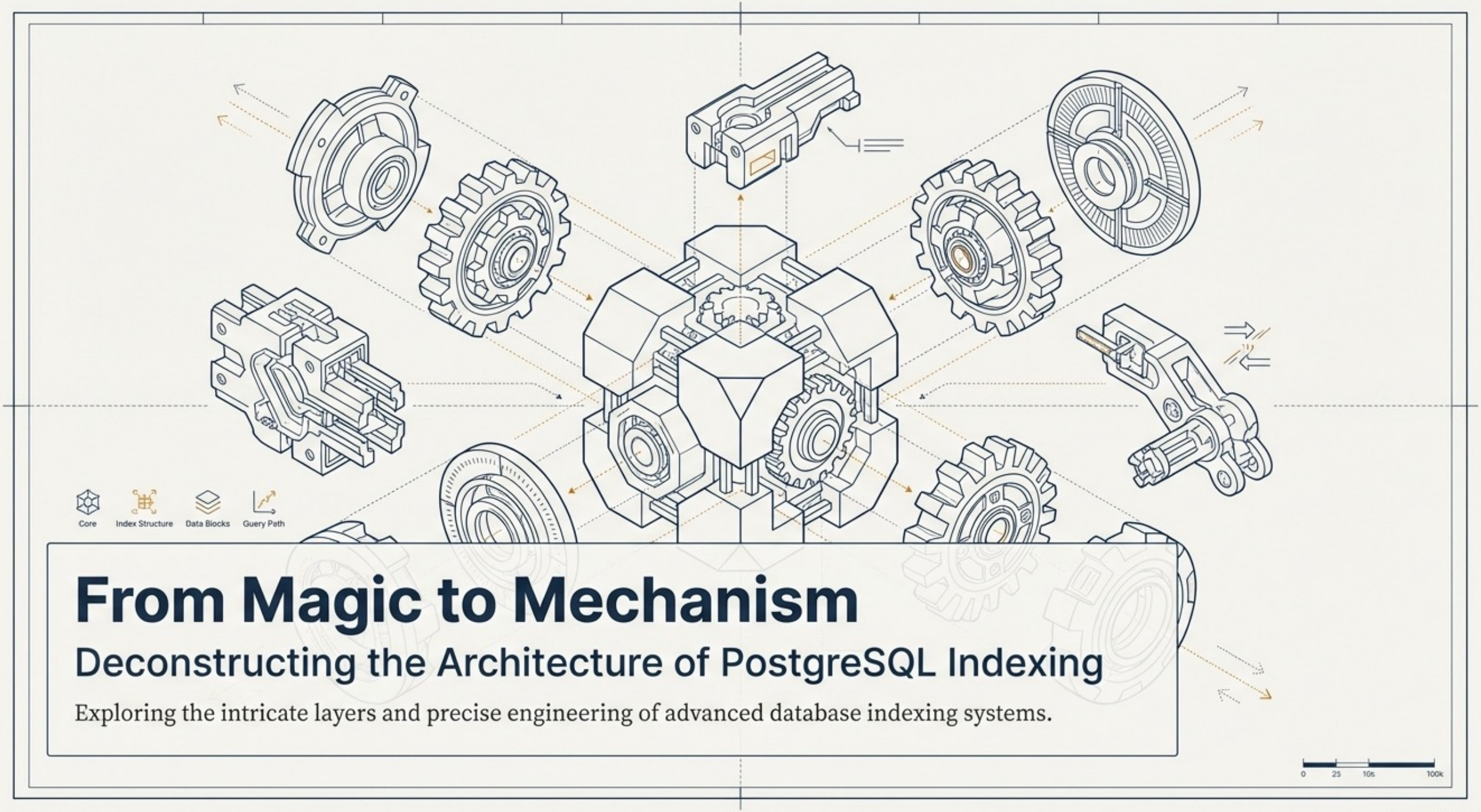
```
max_parallel_workers_per_gather
min_parallel_table_scan_size
PARALLEL UNSAFE/RESTRICTED
```

The Rules of the Game

Master the configuration limits (`..._workers`), table size thresholds, and query restrictions (`PARALLEL UNSAFE/RESTRICTED`) to predict and control parallelism.

NotebookLM

19. Index Access Methods



From Magic to Mechanism

Deconstructing the Architecture of PostgreSQL Indexing

Exploring the intricate layers and precise engineering of advanced database indexing systems.

NotebookLM

The Familiar “Magic” of an Index

We rely on indexes daily to accelerate data access. A sequential scan becomes an index scan, and query times drop from minutes to milliseconds. But this performance magic. It’s the product of an elegant, extensible, and deeply layered architecture. This presentation peels back those layers.

```
EXPLAIN SELECT * FROM tickets
WHERE passenger_name = 'ALEX';

                QUERY PLAN
-----------------------------------
Seq Scan on tickets
  Filter: (passenger_name = 'ALEX')
```



SLOW

```
EXPLAIN SELECT * FROM tickets
WHERE passenger_name = 'ALEX';

                QUERY PLAN
-------------------------------------
Index Scan using tickets_name_idx
  Index Cond: (passenger_name = 'ALEX')
```



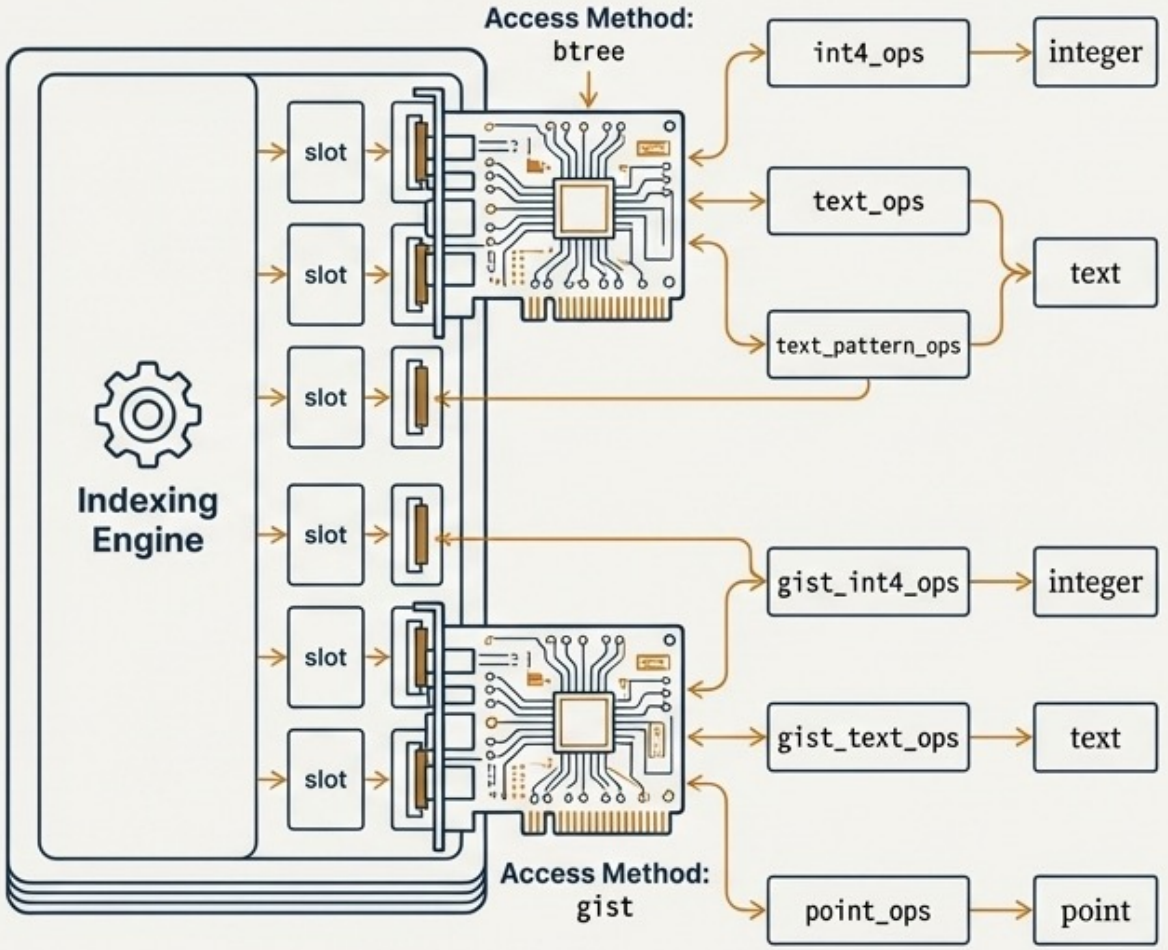
FAST

NotebookLM

The Indexing System: A Pluggable Architecture

PostgreSQL's indexing is not a single entity. It's a modular system with four distinct layers. Understanding their roles and relationships is key to mastering indexing.

- **The Indexing Engine:** The central "motherboard." Handles generic tasks for all index types.
- **Access Methods:** Specialized "hardware cards" (B-Tree, GiST, GIN) that plug into the engine, each providing a unique data structure.
- **Operator Classes:** The "drivers" that teach an Access Method how to handle a specific data type (e.g., how B-Tree handles `text`).
- **Operator Families:** Groups of related "drivers" that enable cross-type comparisons.



NotebookLM

The Indexing Engine is the Conductor

The Engine initiates and processes index scans, but it's agnostic about the specific index type. It provides a common interface, ensuring that no matter which Access Method is used, the core tasks are handled consistently.



Retrieve TIDs
Takes the Tuple IDs (TIDs) returned by an Access Method.



Fetch Heap Tuples
Reads the actual data from the table based on the TIDs.



Check Visibility
Ensures the tuple is visible to the current transaction snapshot (MVCC).



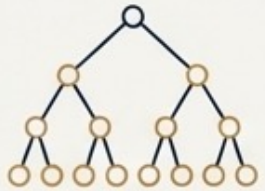
Recheck Conditions
Re-applies query conditions if the index scan was lossy.

The Engine provides the stable foundation, allowing Access Methods to focus on their specialized task: finding TIDs efficiently.

NotebookLM

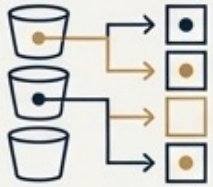
Access Methods are the Specialists

An Access Method is the implementation of an index type. Each uses a different algorithm and data structure, optimized for specific kinds of data and queries. PostgreSQL ships with six core methods, and others can be added as extensions.



B-Tree

General purpose, ordered data.



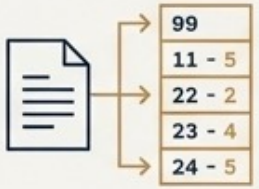
Hash

Simple equality checks.



GiST

Generalized Search Tree (geospatial, full-text).



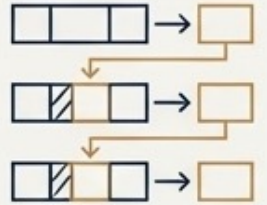
GIN

Generalized Inverted Index (composite values, full-text).



SP-GiST

Space-Partitioned GiST (non-balanced structures).



BRIN

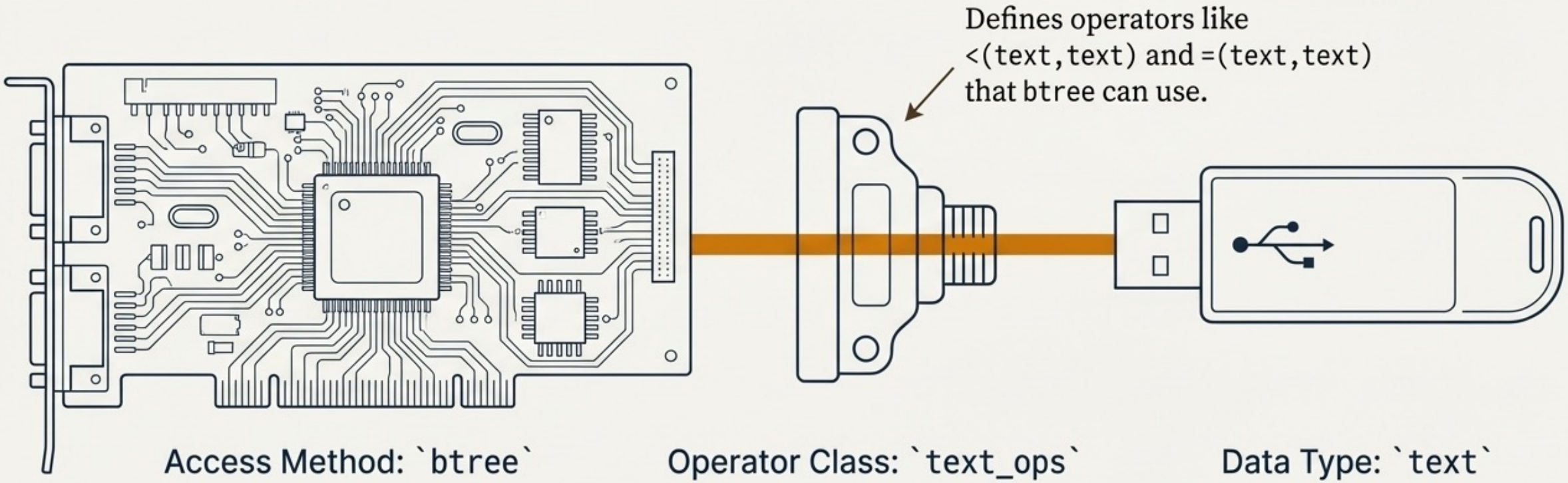
Block Range Index (very large, ordered datasets).

```
=> SELECT amname FROM pg_am WHERE amtype = 'i';
  amname
---------
 btree
 hash
 gist
 gin
 spgist
 brin
(6 rows)
```

NotebookLM

Operator Classes Bridge Methods and Data Types

An Access Method like B-Tree provides a generic algorithm for ordering and comparison, but it doesn't intrinsically know how to compare two `text` strings or two `integer` values. The **Operator Class** provides this knowledge It's a collection of operators and support functions that implements the Access Method's required behaviors for a specific data type.



NotebookLM

A Practical Problem: `LIKE` Queries and Collations

The default B-tree operator class for text, `text_ops`, respects database collations (e.g., `en_US.UTF-8`). This prevents it from being used for prefix `LIKE` searches, as the sorting behavior doesn't match the simple pattern matching required.

```
-- Database using a standard collation
=> SHOW lc_collate;
 lc_collate
-------------
 en_US.UTF-8

-- Index created with the default operator class
=> CREATE INDEX ON tickets(passenger_name);

-- The query cannot use the index
=> EXPLAIN (costs off) SELECT * FROM tickets
      WHERE passenger_name LIKE 'ELENA%';

                QUERY PLAN
-----------------------------------------
 Seq Scan on tickets
   Filter: (passenger_name ~~ 'ELENA%')
(2 rows)
```

The index is ignored because `text_ops` cannot satisfy this `LIKE` condition with the current collation.

NotebookLM

The Solution: `text_pattern_ops`

PostgreSQL provides an alternative operator class, `text_pattern_ops`. Its comparison operators perform simple, byte-wise comparisons, ignoring collation. This makes it suitable for pattern matching operators like `LIKE` and `~`.

```
-- Create an index specifying the non-default operator class
=> CREATE INDEX ON tickets(passenger_name text_pattern_ops);

-- The same query now uses an index scan
=> EXPLAIN (costs off) SELECT * FROM tickets
     WHERE passenger_name LIKE 'ELENA%';

                      QUERY PLAN
-----------------------------------------------------
 Bitmap Heap Scan on tickets
   -> Bitmap Index Scan on tickets_..._pattern_idx
        Index Cond: ((passenger_name ~>=~ 'ELENA') AND
                     (passenger_name ~<~ 'ELENB'))
```

The planner intelligently rewrote the `LIKE` into a range scan using operators from `text_pattern_ops` that ignore collation.

NotebookLM

Strategies: The Language of Access Methods

An Operator Class maps human-readable operators to abstract **strategy numbers** that the Access Method understands. For B-Tree, strategy `1` always means 'less than' and `3` always means 'equal,' regardless of the data type or operator symbol.

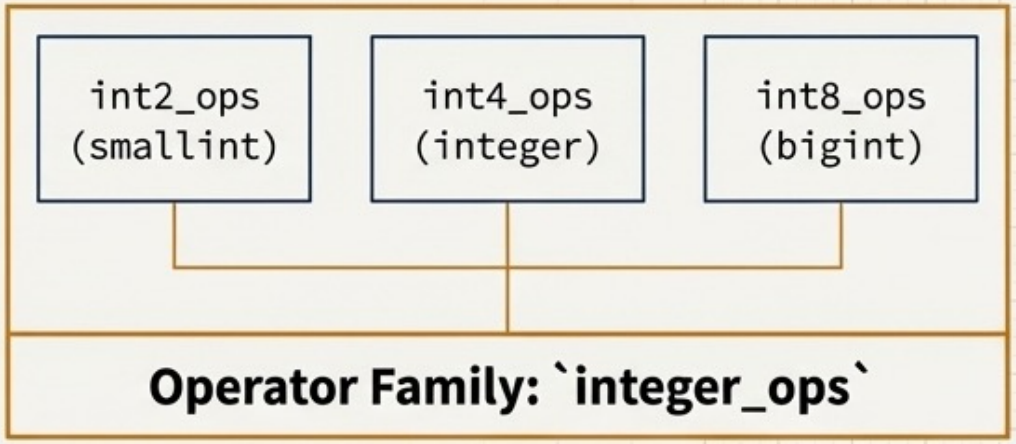
opcname	amopstrategy	amopopr (Operator)
text_ops	1	<(text,text)
	3	=(text,text)
	5	>(text,text)
text_pattern_ops		
	1	~<~(text,text)
	3	=(text,text)
	5	~>~(text,text)

This mapping allows the `btree` method to remain generic, while operator classes provide the specific implementation details for each data type.

NotebookLM

Operator Families Enable Cross-Type Operations

An **Operator Family** groups together operator classes that handle similar data types with compatible semantics. For example, the `integer_ops` family contains classes for `smallint`, `integer`, and `bigint`. By grouping these, the query planner can use an index on an `integer` column even when the comparison value is a `bigint`, without needing an explicit cast. This improves performance and query flexibility.




```
-- The `integer_ops` family contains classes for
multiple integer types
=> SELECT opcname, opcintype::regtype
   FROM pg_opfamily opf ...
   WHERE opfname = 'integer_ops';

  opcname  | opcintype
-----------+-----------
  int2_ops | smallint
  int4_ops | integer
  int8_ops | bigint
```

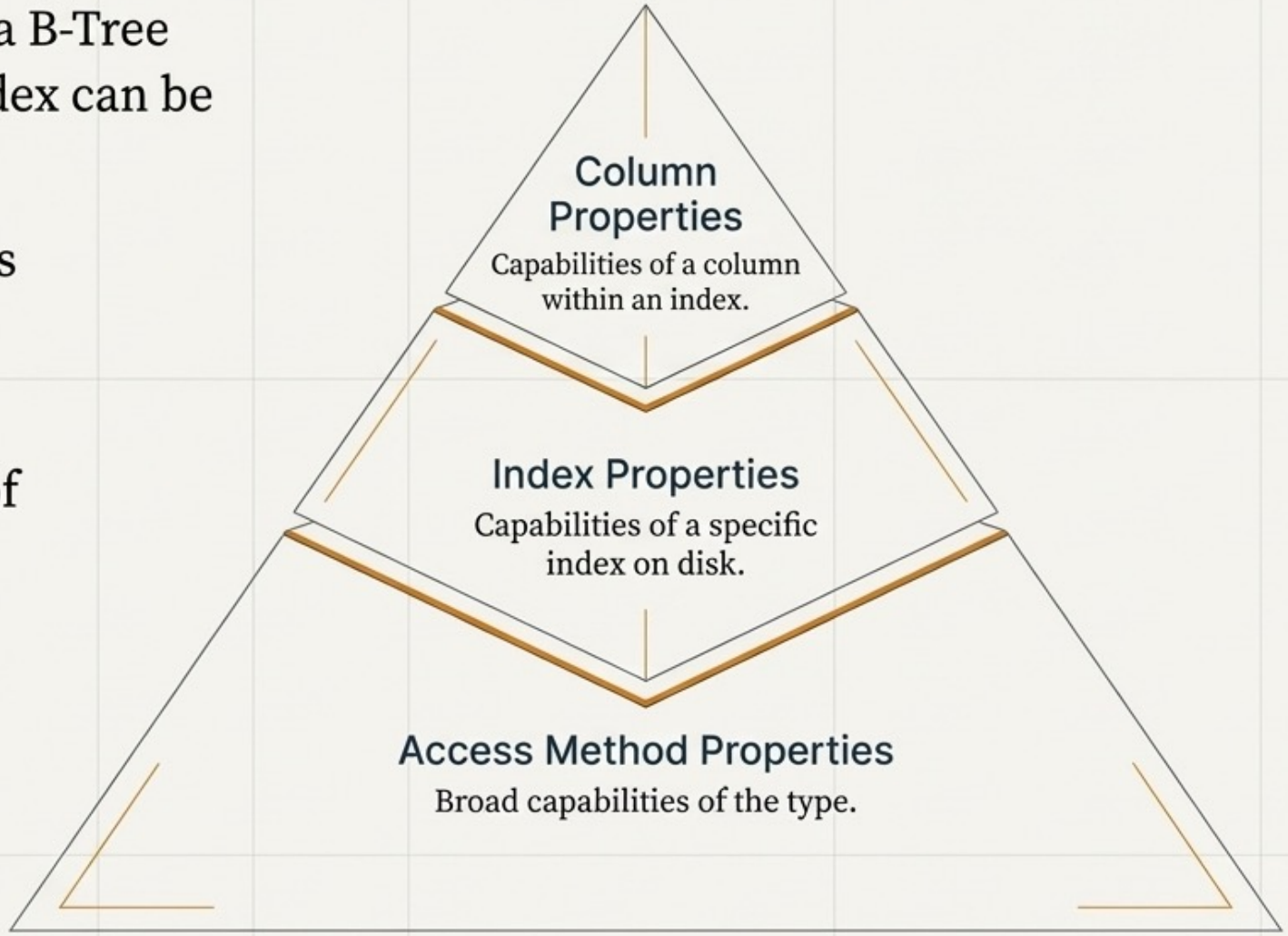
NotebookLM

The Contract: How the Planner Uses Index Properties

How does the query planner know if a B-Tree can return sorted data or if a GiST index can be used for an Index-Only Scan?

It queries the properties of the access method, index, and columns.

These boolean flags act as a formal contract, declaring the capabilities of each component.



NotebookLM

Access Method Properties: The Spec Sheet

These five properties define the fundamental capabilities of an access method. B-Tree, being the most mature, supports all five.



can_order Returns data in sorted order, avoiding a separate `Sort` step.



can_unique Can enforce `UNIQUE` and `PRIMARY KEY` constraints.



can_multi_col Can be built on multiple columns.



can_exclude Can enforce `EXCLUDE` constraints (for generalized uniqueness).



can_include Can include non-key columns for covering indexes.

```
=> SELECT name, pg_indexam_has_property('btree', name)
     FROM unnest(array['can_order', 'can_unique', ...]);

    name     | pg_indexam_has_property
-------------+-------------------------
 can_order   | t
 can_unique  | t
 ...         | ...
```

NotebookLM

Index and Column Properties: Fine-Grained Capabilities

These properties describe the behavior of a specific index instance or a column within it, giving the planner even more detailed information for optimization.

Index-Level Properties

clusterable
Supports the `CLUSTER` command.

index_scan
Can return TIDs one by one.

bitmap_scan
Can return a bitmap of all matching TIDs at once.

backward_scan
Can be scanned in reverse order.

Key Column-Level Properties

returnable
The indexed value can be returned directly from the index, enabling **Index-Only Scans**.

distance_orderable
Supports ordering by a 'distance' operator (e.g., `point <-> point` in GiST for k-NN searches).

search_array
Supports `ANY (array)` conditions, which the planner uses for `IN (...)` lists.

search_nulls
Can efficiently search for `IS NULL` or `IS NOT NULL`.

NotebookLM

The Full Mechanism: From Query to Result

Let's trace the path. When a query arrives, the planner uses this architecture to find the optimal path.

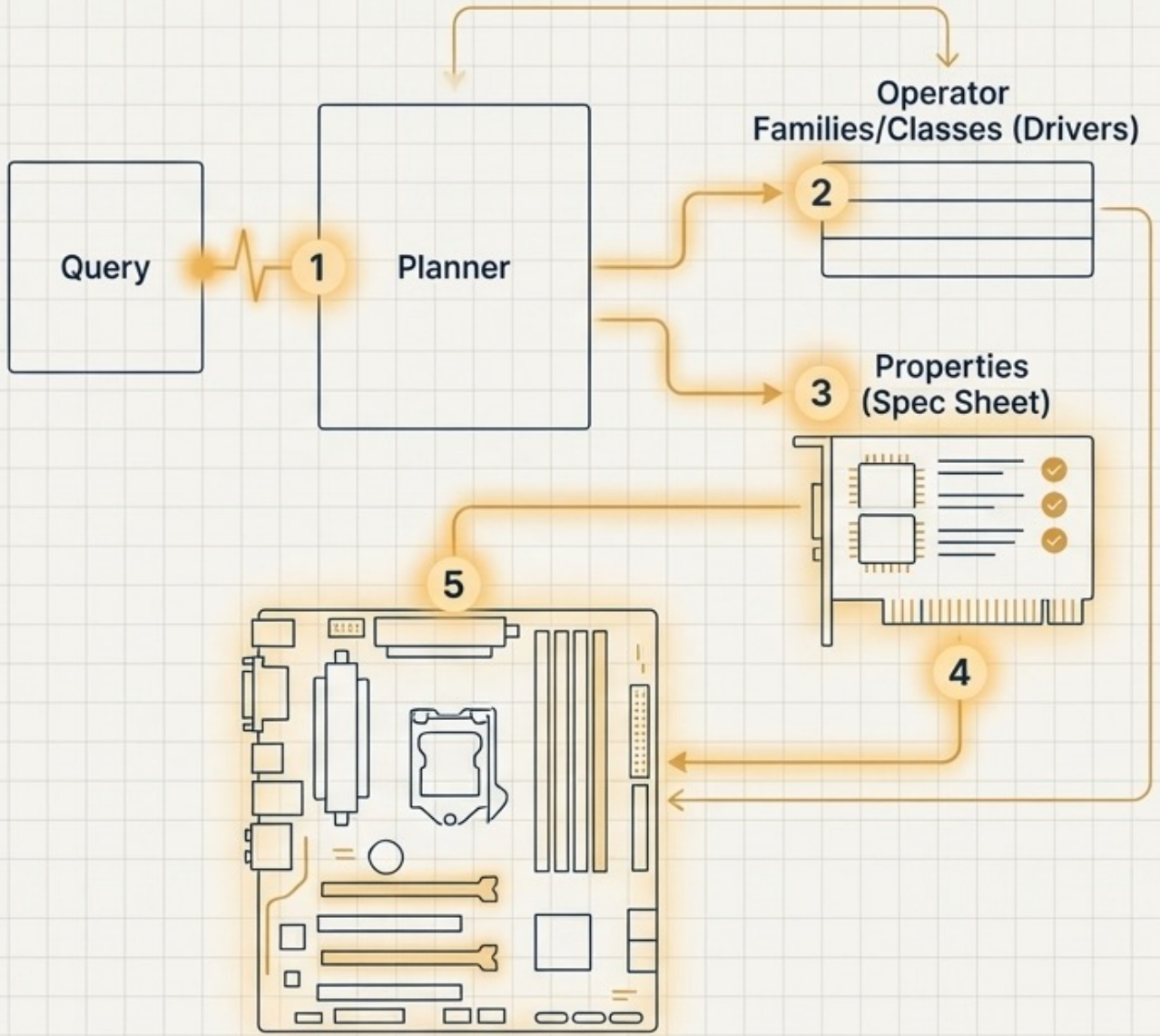
1. **Planner**
   Examines the query's operators (`=`, `LIKE`, `<->`).

2. **Operator Families/Classes (Drivers)**
   Finds a matching operator and identifies the corresponding Access Method.

3. **Properties (Spec Sheet)**
   Checks the AM and index properties to confirm capabilities (e.g., 'Can this index return sorted data?').

4. **Access Method (Interface/Hardware Card)**
   If viable, the planner instructs the AM (via its handler functions) to begin a scan.

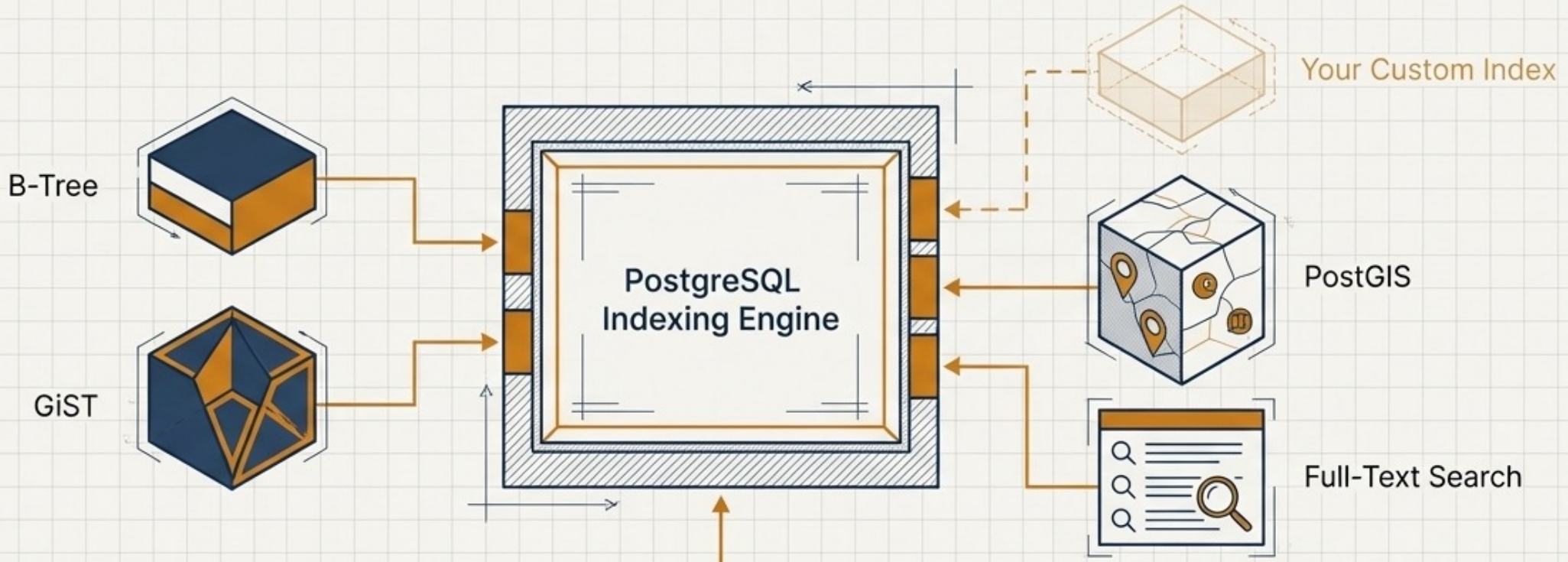
5. **Indexing Engine (Kernel)**
   The engine receives the TIDs from the AM, fetches the data from the heap, and returns the final result.



NotebookLM

The Architecture is the Feature

The true power of PostgreSQL's indexing lies lies not in any single Access Method, but in its extensible design. This layered architecture separates the generic from the specific, allowing the database to evolve. New data data structures, data types, and query behaviors can be integrated without altering the core engine.



Understanding this mechanism empowers you to move beyond default indexes, choosing and even creating the right tools to solve complex data challenges.

NotebookLM

20. Index Scans

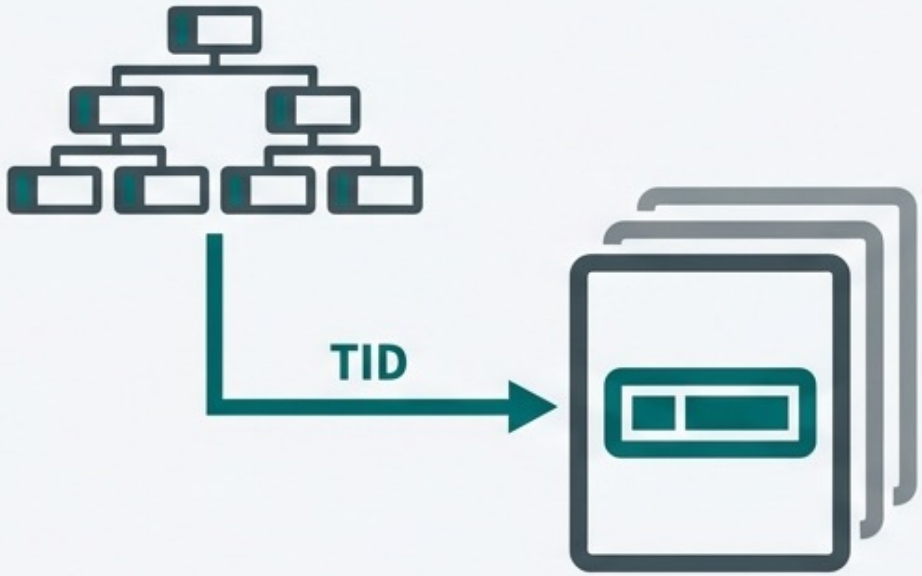
The Planner’s Dilemma: How PostgreSQL Chooses the Right Index Scan

An inside look at the cost-based decisions behind Index Scans, Bitmap Scans, and Index-Only Scans.

NotebookLM

The Default Tool: The Index Scan

The most direct way to use an index. The planner uses the index to find the location (TID) of a row, then fetches the row from the main table (the heap).




```
EXPLAIN SELECT * FROM bookings
WHERE book_ref = '9AC0C6';

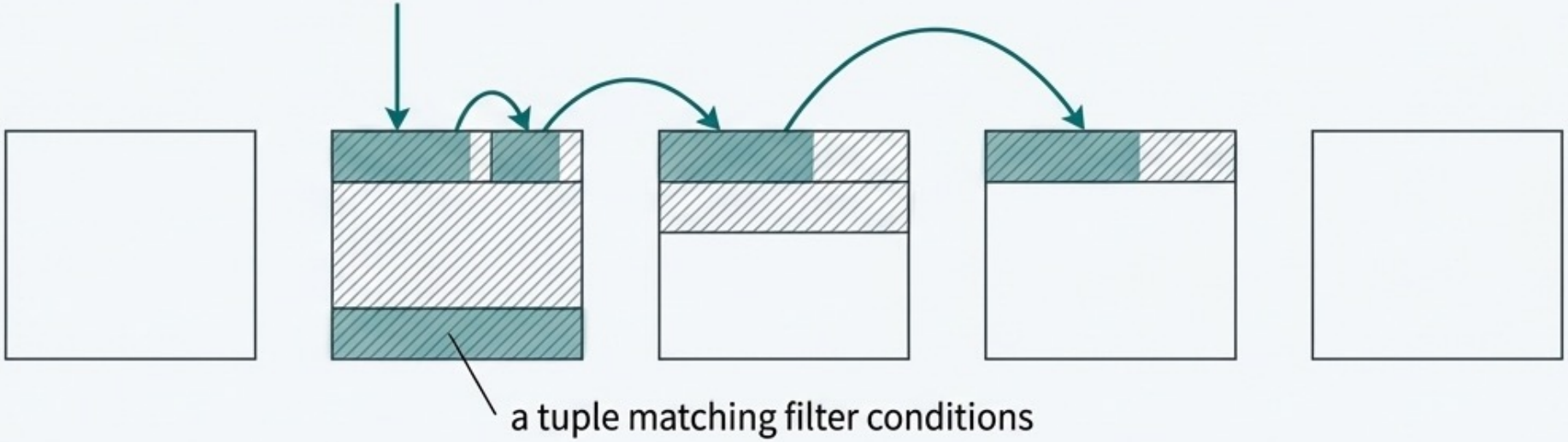
-- QUERY PLAN ------------------------------
-- Index Scan using bookings_pkey on bookings
--   (cost=0.43..8.45 rows=1 width=21)
--   Index Cond: (book_ref = '9AC0C6'::bpchar)
```

Key Takeaway: The `Index Scan` node handles both finding the Tuple ID (TID) in the index and fetching the tuple from the heap. `Index Cond` shows the condition pushed down to the index.

NotebookLM

The Ideal Case: High Correlation

When the physical order of data on disk closely matches the logical order in the index, performance is optimal. The planner can read heap pages sequentially, which is fast.



- PostgreSQL collects statistics on the correlation between physical and logical row order. A value close to 1 (or -1) indicates high correlation.
- With high correlation, the **Index Scan** reads one heap page after another, minimizing random I/O.

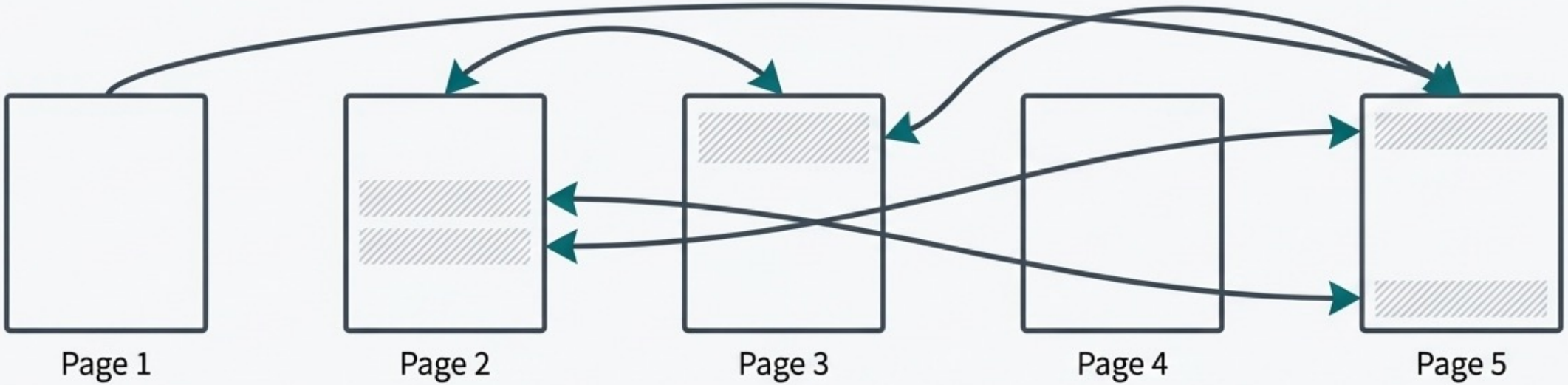
```
SELECT attname, correlation FROM pg_stats
WHERE tablename = 'bookings' AND attname = 'book_ref';
```

```
  attname   | correlation
------------+-------------
 book_ref |           1
```

NotebookLM

The Realistic Challenge: Low Correlation

When data is not physically ordered according to the index (low correlation), the planner must jump between random heap pages. This random I/O is slow and expensive.



- Low correlation forces the `Index Scan` to perform many random disk reads, which can be orders of magnitude slower than sequential reads.
- In the worst case, the number of page accesses can approach the number of rows fetched.

```
-- Forcing an Index Scan on a low-correlation column
EXPLAIN SELECT * FROM bookings
WHERE book_date < '2016-08-23 12:00:00+03';

-- QUERY PLAN ----------------------------------------
-- Index Scan using bookings_book_date_idx on bookings
--   (cost=0.43..56957.48 rows=132403 width=21)
```

NotebookLM

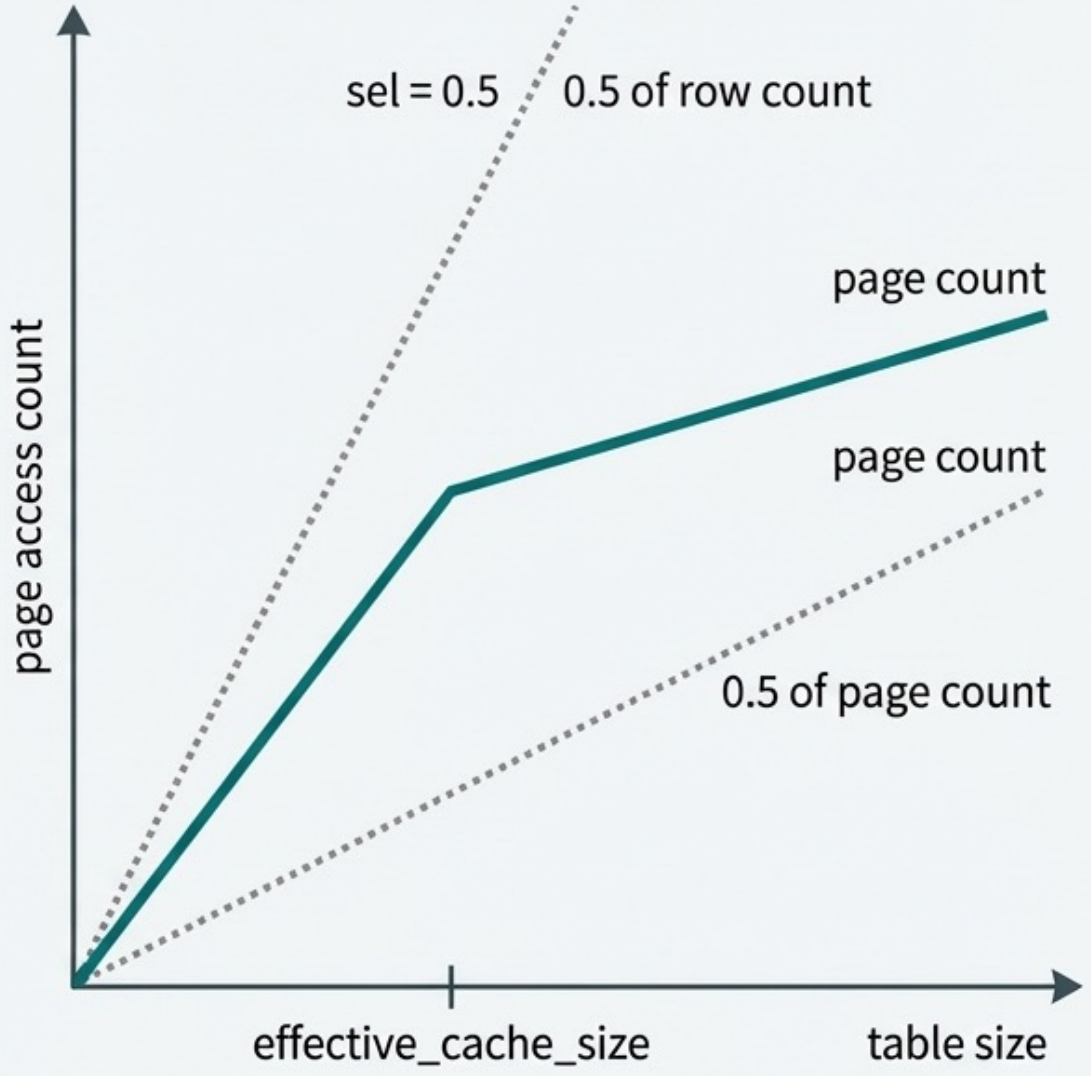
The Planner's Hedge: The Role of Caching

The planner doesn't assume every read comes from disk. It models the effect of the buffer cache using the `effective_cache_size` parameter to estimate the true cost of random I/O.

The planner assumes that frequently accessed pages will be in the cache, avoiding a physical disk read.

A larger `effective_cache_size` tells the planner that more of the table can fit in memory, lowering the estimated cost of random-access scans.

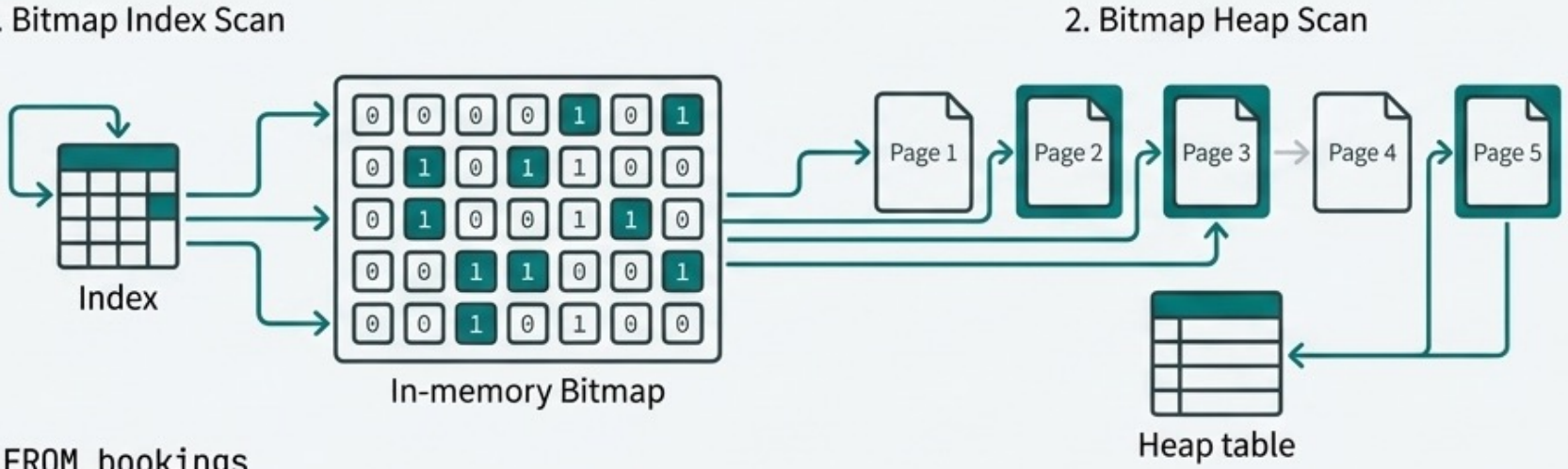
The dotted lines represent the theoretical best case (perfect correlation) and worst case (zero correlation, no cache). The solid line shows the planner's more realistic estimate.



NotebookLM

An Alternative for Chaos: The Bitmap Scan

Think of it as making a shopping list. Instead of running back and forth across the store for each item, you first identify all the aisle numbers you need, sort them, and then visit each aisle only once.



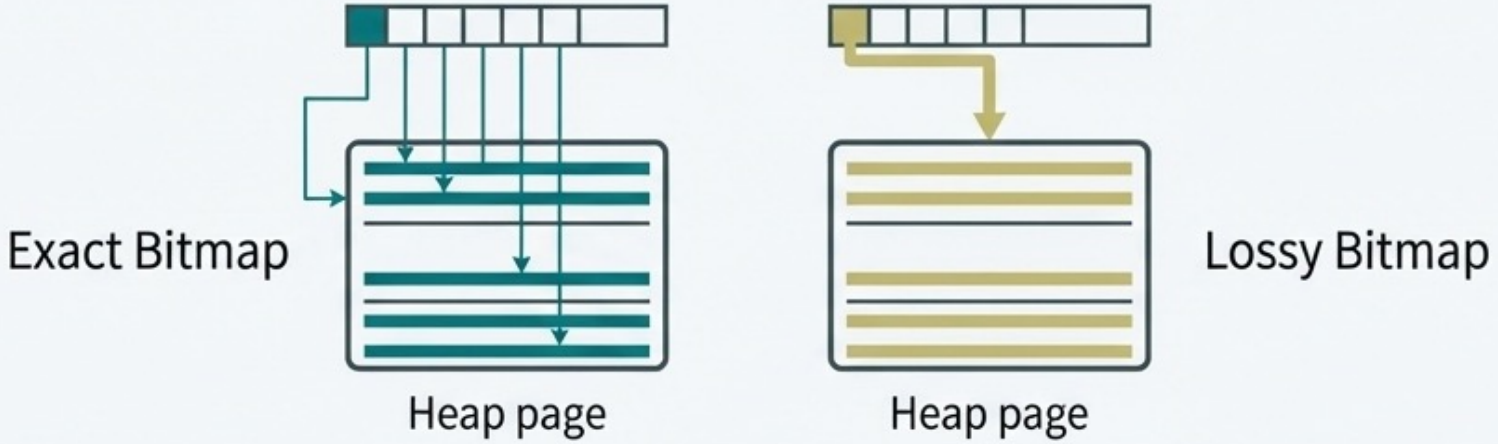

```
EXPLAIN SELECT * FROM bookings
WHERE total_amount = 48500.00;

-- QUERY PLAN ------------------------------------------
-- Bitmap Heap Scan on bookings
--   Recheck Cond: (total_amount = 48500.00)
--   -> Bitmap Index Scan on bookings_total_amount_idx
--        Index Cond: (total_amount = 48500.00)
```

NotebookLM

A Trade-Off: Bitmap Memory vs. Accuracy

Bitmaps are built in memory, limited by `work_mem`. If the bitmap grows too large, PostgreSQL makes it “lossy” to save space, at the cost of re-checking more rows.



Sufficient `work_mem`

```
EXPLAIN (ANALYZE, ...)
SELECT * FROM bookings WHERE total_amount > 150000;

-- ...
-- Heap Blocks: exact=13447
-- ...
```

Reduced `work_mem`

```
SET work_mem = '512kB';
EXPLAIN (ANALYZE, ...)
SELECT * FROM bookings WHERE total_amount > 150000;

-- ...
-- Rows Removed by Index Recheck: 1145721
-- Heap Blocks: exact=5178 lossy=8269
-- ...
```

A “lossy” block forces the planner to re-check the condition for every tuple on that page, as shown by `Rows Removed by Index Recheck`.

NotebookLM

The Power of Combination: Merging Multiple Indexes

Bitmap Scans can combine results from multiple indexes on the same table using AND and OR logic before ever touching the table itself.

The planner creates a bitmap from each index scan and then merges them using bitwise operations (BitmapAnd, BitmapOr). This can dramatically reduce the number of rows that need to be fetched from the heap.

```
EXPLAIN SELECT * FROM bookings
WHERE book_date < '2016-08-28' AND total_amount > 250000;

-- QUERY PLAN ----------------------------------------
-- Bitmap Heap Scan on bookings
--   -> BitmapAnd
--        -> Bitmap Index Scan on bookings_total_amount_idx
--             Index Cond: (total_amount > '250000'::numeric)
--        -> Bitmap Index Scan on bookings_book_date_idx
--             Index Cond: (book_date < '2016-08-28'::...)
```

Secondary font Source Sans Pro

Code in JetBrains Mono

NotebookLM

The Ultimate Shortcut: The Index-Only Scan

The “Cheat Sheet.” Getting the answer directly from the index without ever needing to open the textbook (the heap).

If all columns required by the query are available within the index itself (a “covering index”), PostgreSQL can avoid accessing the main table, leading to a massive performance gain.



```
EXPLAIN SELECT book_ref FROM bookings
WHERE book_ref < '100000';

-- QUERY PLAN ---------------------------------
-- Index Only Scan using bookings_pkey on bookings
--   (cost=0.43..3791.91 rows=132999 width=7)
--   Index Cond: (book_ref < '100000'::bpchar)
```

Explain that `INCLUDE` clauses can be used to create covering indexes without affecting uniqueness constraints.

NotebookLM

The Fine Print: Visibility and `VACUUM`

An Index-Only Scan isn't truly "index only." PostgreSQL must still verify that the rows are visible to the current transaction. This is optimized using a special data structure called the Visibility Map.

The Visibility Map tracks pages where every tuple is visible to all transactions. If a tuple's page is marked in the map, its visibility check can be skipped, avoiding a heap fetch.
The `VACUUM` process is responsible for updating the Visibility Map.

Before VACUUM

```
-- On a newly loaded table
EXPLAIN (ANALYZE, ...)
SELECT book_ref FROM bookings_tmp WHERE ...;

-- Index Only Scan on bookings_tmp ...
--   Heap Fetches: 132109
```

After VACUUM

```
VACUUM bookings_tmp;
EXPLAIN (ANALYZE, ...)
SELECT book_ref FROM bookings_tmp WHERE ...;

-- Index Only Scan on bookings_tmp ...
--   Heap Fetches: 0
```

Key Takeaway: Without regular vacuuming, an Index-Only Scan can degrade to the performance of a regular Index Scan.

NotebookLM

Scaling Up: Parallel Execution

All three major **index scan** methods—Index Scan, Index-Only Scan, and Bitmap Scan—have parallel variants to distribute the CPU workload across multiple workers.

- The planner can choose a parallel plan to reduce query time.
- I/O costs are generally not split, as page access is coordinated. CPU costs, however, are divided among workers.
- In a Bitmap Scan, the `Bitmap Index Scan` (building the map) is performed by a single leader, while the `Bitmap Heap Scan` (reading the table) is parallelized.

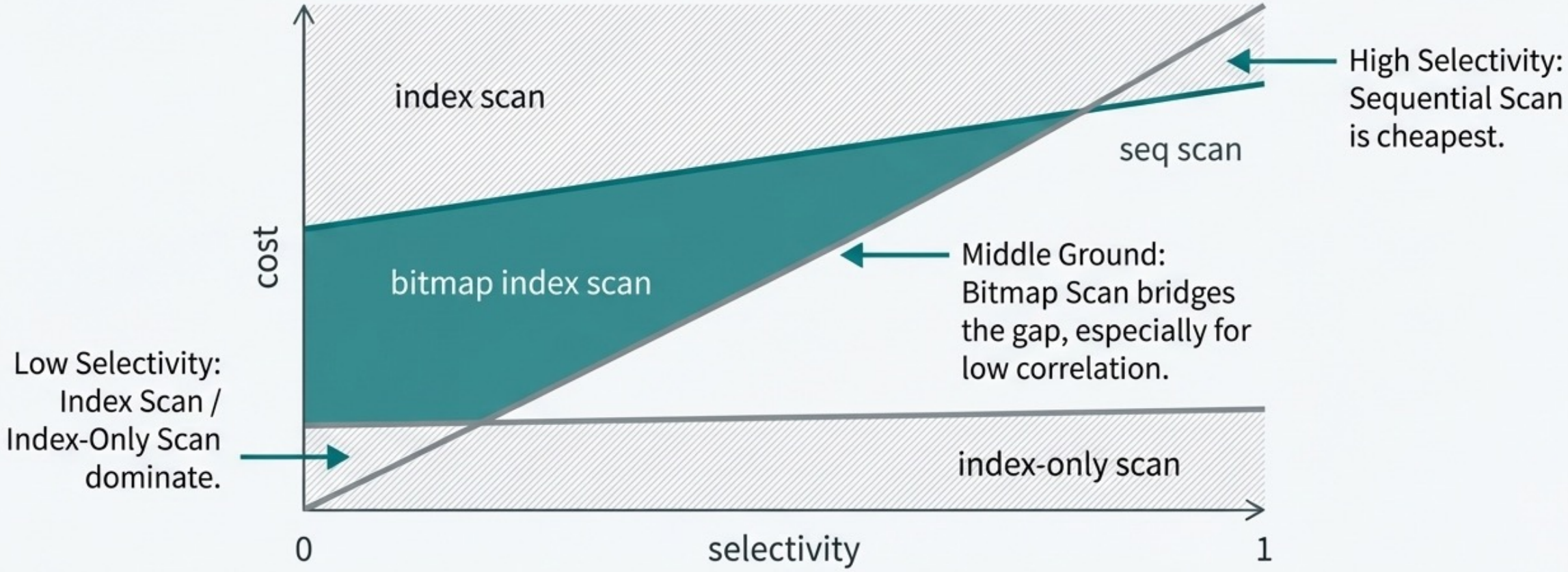
```
EXPLAIN SELECT sum(total_amount) FROM bookings
WHERE book_ref < '400000';

-- QUERY PLAN ----------------------------------------
-- ...
--  -> Gather (cost=19192.59..19192.80 rows=2 width=32)
--       Workers Planned: 2
--       -> Partial Aggregate ...
--          -> Parallel Index Scan using bookings_pkey
```

NotebookLM

The Planner's Verdict: A Visual Comparison

- This graph synthesizes the cost estimates for each access method based on query selectivity (the percentage of rows returned).



NotebookLM

Deconstructing the Decision

Each access method has a scenario where it excels. The planner's job is to calculate the costs and pick the winner for your specific query.



- **Sequential Scan**
  Cost is constant regardless of selectivity. Becomes the best choice when a large percentage of the table must be read.



- **Index Scan**
  The undisputed champion for fetching a very small number of rows. Its cost is heavily dependent on correlation; low correlation makes it expensive quickly.



- **Index-Only Scan**
  Extremely fast when applicable (covering index, all-visible pages). Can outperform even a Sequential Scan for high-selectivity queries, but degrades to an Index Scan if heap fetches are required.



- **Bitmap Scan**
  The versatile middle ground. Outperforms a low-correlation Index Scan as selectivity grows. Less sensitive to correlation but more sensitive to work_mem.

NotebookLM

The Planner’s Checklist

The planner's decision is a multi-factor equation. Understanding these inputs is key to performance tuning.



Selectivity

How many rows does the query need? The most fundamental input.



Correlation

Are the rows physically ordered like the index? Crucial for `Index Scan` cost.



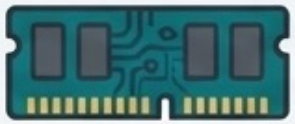
Column Coverage

Are all needed columns in the index? Enables `Index-Only Scan`.



Visibility

Are the pages ‘all-visible’? Determines `Index-Only Scan` efficiency.



Available Memory

`effective_cache_size` influences random I/O cost; `work_mem` affects `Bitmap Scan` accuracy.



System Costs

The configured costs for CPU and I/O operations (`seq_page_cost`, `random_page_cost`, etc.).

NotebookLM

There Is No 'Best' Method, Only the Right Tool for the Job

The planner performs extensive calculations to choose the most efficient path for every query. Its intelligence, however, is entirely dependent on the accuracy of the statistics it is given.

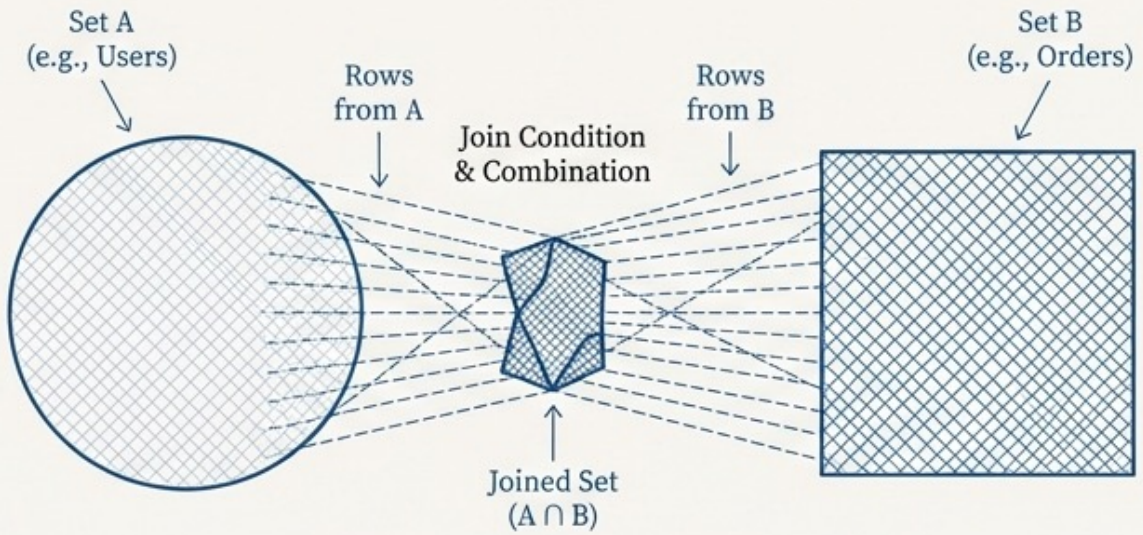
Your role is not to outsmart the planner, but to ensure it has the accurate data it needs to make the best decision. Analyze your tables.

NotebookLM

21. Nested Loop

Joins are the Core of Relational Databases

Before exploring how PostgreSQL executes a join, we must first define what a join is. Joins are the cornerstone of SQL's expressive power, providing the logical framework for combining distinct sets of rows.

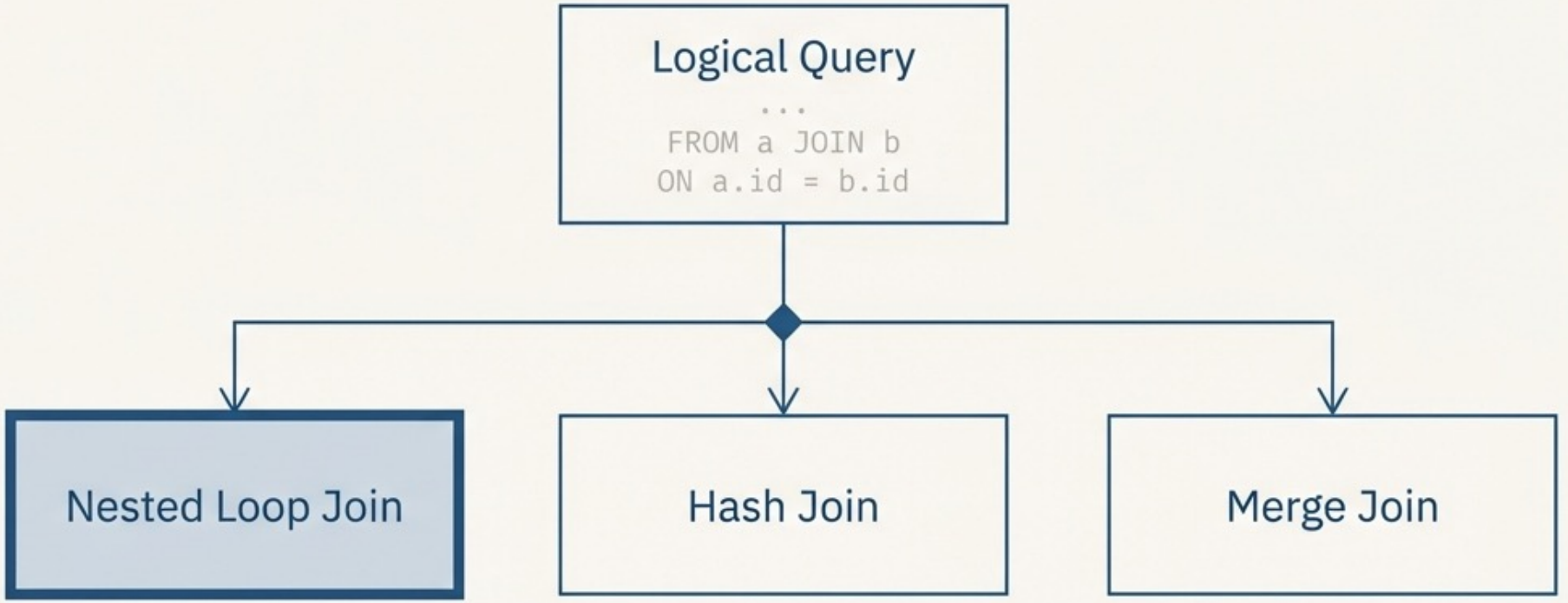


- **Inner Join:** Returns only the rows that satisfy the join condition. The most common type is the Equi-join, based on equality. A CROSS JOIN is a special case of an inner join with a condition that is always true, resulting in a Cartesian product.
- **Outer Join:** Extends the result of an Inner Join. A LEFT JOIN includes non-matching rows from the left set, a RIGHT JOIN from the right set, and a FULL JOIN from both. Gaps in the non-matching side are filled with NULL values.
- **Semi Join:** Returns rows from the left set if and only if a match exists in the right set. Crucially, it does not duplicate left-side rows, even if multiple matches exist on the right.
- **Anti Join:** The opposite of a Semi Join. It returns rows from the left set if and only if no match exists in the right set. SQL implements Semi and Anti joins using the EXISTS and NOT EXISTS predicates, respectively.

NotebookLM

PostgreSQL's Three Methods for Executing Joins

A logical join operation (e.g., `INNER JOIN`) must be implemented by a physical join *method*. The query planner's core responsibility is to analyze the query, the data statistics, and the available indexes to select the most cost-effective physical algorithm for any given situation.



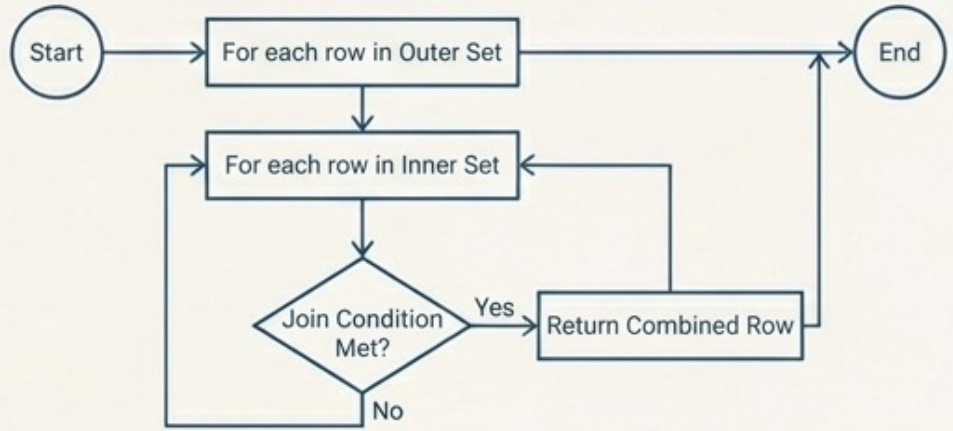
This deep-dive focuses on the **Nested Loop Join**, a foundational and versatile algorithm that can physically implement Inner, Left Outer, Semi, and Anti joins.

NotebookLM

The Core Algorithm: A Loop Within a Loop

The fundamental Nested Loop algorithm is perfectly described by its name. It is the most intuitive approach to joining two sets of data:

1. The **outer loop** iterates through every row of the first table (the 'outer set').
2. For *each* of those rows, the **inner loop** scans the entire second table (the 'inner set') to find rows that satisfy the join condition.
3. Matching pairs are yielded and returned immediately.




```
FOR each row_outer IN outer_table:
  FOR each row_inner IN inner_table:
    IF join_condition_met(row_outer, row_inner):
      RETURN  combined_row
```

What Determines Its Efficiency?

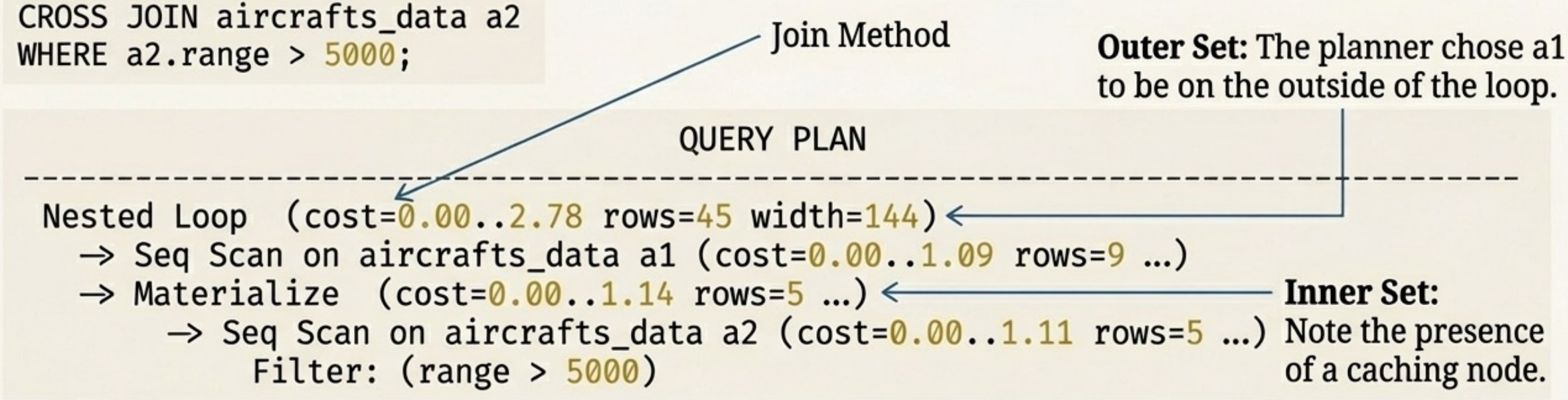
- The cardinality (total number of rows) of the outer set, which dictates the number of inner loop executions.
- The ability to efficiently retrieve matching rows from the inner set (e.g., using an index vs. a full table scan).
- The cost of repeatedly accessing the same data in the inner set if the outer set contains non-unique join keys.

NotebookLM

The Nested Loop in Action: The Cartesian Product

SQL Query

```
EXPLAIN SELECT *
FROM aircrafts_data a1
CROSS JOIN aircrafts_data a2
WHERE a2.range > 5000;
```



The `Nested Loop` node always has two children: the upper child is the outer set, and the lower is the inner set. For each of the 9 rows from the outer scan (a1), the planner will access the inner set.

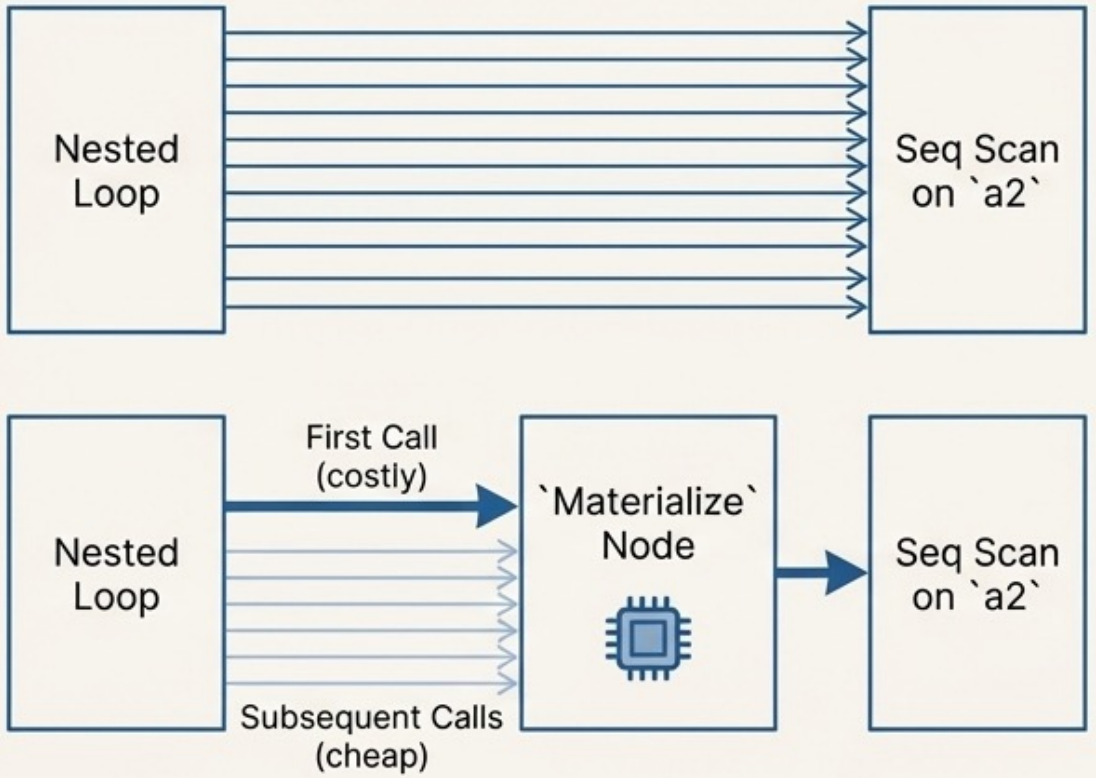
But notice the `Materialize` node. PostgreSQL is already performing a subtle optimization to avoid the cost of re-scanning and re-filtering the `aircrafts_data` table 9 separate times. Let's deconstruct how.

NotebookLM

First Optimization: The `Materialize` Node Caches the Inner Set

Anatomy of the `Materialize` Node

- **Purpose:** To avoid the expensive re-computation or re-scanning of its child node (the inner set).
- **Mechanism:** On its first call from the parent `Nested Loop`, it executes its child plan (e.g., a `Seq Scan` with a filter) and stores all resulting rows in an in-memory buffer.
- **Subsequent Calls:** On every subsequent call, it returns the stored rows directly from the memory buffer, completely bypassing its child node. This is significantly cheaper than re-executing the scan.
- **Memory Management:** It accumulates rows in memory up to the `work_mem` limit (default 4MB). If this limit is exceeded, the data is spilled to a temporary file on disk, which is still faster than re-scanning the original table.



Impact: This dramatically reduces the **total cost** of the **join** by paying the price of scanning the inner table only once, then relying on cheap memory/disk reads for all subsequent iterations.

NotebookLM

Deconstructing the Cost with `Materialize`

```
                          QUERY PLAN
-----------------------------------------------------------
 Nested Loop  (cost=0.00..2.78 rows=45 width=144)
   -> Seq Scan on aircrafts_data a1
        (cost=0.00..1.09 rows=9 width=72)
   -> Materialize                          ×9
        (cost=0.00..1.14 rows=5 width=72)
        -> Seq Scan on aircrafts_data a2
             (cost=0.00..1.11 rows=5 width=72)
             Filter: (range > 5000)
```

Cost Components

1. **Outer Set Cost** : Cost to get all 9 rows from the outer set. (`1.09`)
2. **Inner Set First Run**: Cost to scan, filter, and materialize the inner set *once*. (`1.14`)
3. **Inner Set Rescans:** Cost to read the materialized data for the remaining 8 outer rows. This rescan cost is a low internal planner constant (`8 * 0.0125`).
4. **CPU Tuple Cost**: A small CPU processing cost for each of the 45 final rows produced by the join.

The Math

```
-- The planner's calculation can be replicated:
SELECT 0.00 + 0.00 AS startup_cost,
       round((
          1.09 +                                          -- 1. Outer set cost
          (1.14 + 8 * 0.0125) +                           -- 2 & 3. Inner set total cost
          45 * current_setting('cpu_tuple_cost')::real -- 4. CPU cost
       )::numeric, 2) AS total_cost;

-- Result: total_cost = 2.78
```

NotebookLM

Adding Complexity: The Parameterized Index Scan

We now move from the simple CROSS JOIN to a far more common and powerful pattern. Instead of scanning the entire inner table, we will perform a highly specific, indexed lookup for *each row* from the outer table.

```
EXPLAIN SELECT *
FROM tickets t
JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
WHERE t.book_ref = '03A76D';

QUERY PLAN
------------------------------------------------------------------
 Nested Loop  (cost=0.99..45.68 rows=6 width=136)
   -> Index Scan using tickets_book_ref_idx on tickets t
      (cost=0.43..12.46 rows=2 width=104)
       Index Cond: (book_ref = '03A76D'::bpchar)
   ->  Index Scan using ticket_flights_pkey on ticket_flights tf
       (cost=0.56..16.58 rows=3 width=32)
       Index Cond: (ticket_no = t.ticket_no)
```

Key Insight: This is not a static value. The value for `t.ticket_no` is passed down from the outer loop as a **parameter** for each execution of this inner index scan.

- The outer Index Scan finds 2 tickets for the given book_ref.
- For *each* of those two tickets, the Nested Loop node executes the inner Index Scan.
- This parameterization is what makes the Nested Loop so effective for joins with high selectivity on the inner side. Instead of scanning a whole table, it can perform two very fast, targeted index lookups.

NotebookLM

Anatomy of Cardinality Estimation

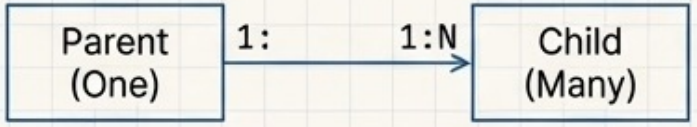
How does the planner estimate the final result will be 6 rows? It multiplies the cardinalities of the two inputs by the calculated **join selectivity**.

```
Estimated Rows = (Outer Rows) * (Inner Rows) * (Selectivity)
```

How Selectivity is Estimated

1. Best Case (Foreign Keys)

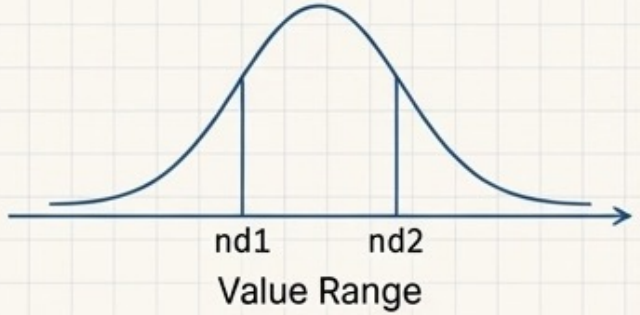
If a foreign key constraint exists between the join keys, the planner has perfect information: it knows each row in the "many" table has exactly one corresponding row in the "one" table. The **select**ivity is simply `1 / (number of rows in the parent table)`. This is the most accurate method.




```
-- Estimated Rows = 2 * (tf.reltuples * (1.0 / t.reltuples))
SELECT round(2 * tf.reltuples * (1.0 / t.reltuples)) AS rows
FROM pg_class t, pg_class tf
WHERE t.relname = 'tickets' AND tf.relname = 'ticket_flights';
-- rows: 6
```

2. Fallback (No Foreign Key)

Without a foreign key, the planner must rely on table statistics. For a standard equi-join, it uses the number of distinct values (`**n_distinct**`) for the join key in each table. The formula is `min(1/nd1, 1/nd2)`. This statistical approach is powerful but can be less accurate than using a defined constraint.

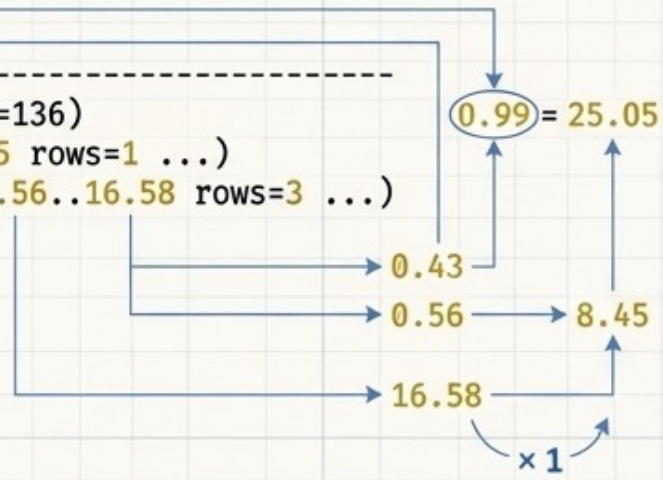


Key Takeaway: Defining foreign keys is not just for data integrity. It provides the query planner with crucial information to generate more accurate, and therefore more efficient, query plans.

NotebookLM

Deconstructing the Cost of a Parameterized Join

```
QUERY PLAN
------------------------------------------------------------------------
 Nested Loop  (cost=0.99..25.05 rows=3 width=136)
  -> Index Scan on tickets t (cost=0.43..8.45 rows=1 ...)
  -> Index Scan on ticket_flights tf (cost=0.56..16.58 rows=3 ...)
```



Cost Components

1. **Startup Cost:** The sum of the startup costs of both child nodes (0.43 + 0.56 = 0.99).
2. **Outer Set Cost:** The total cost to retrieve all rows from the outer scan (8.45).
3. **Inner Set Executions:** The total cost of the inner scan (16.57) is paid for *each row* returned by the outer scan. Here, 1 * 16.57.
4. **CPU Tuple Cost:** The CPU cost for processing the 3 resulting rows.

The Math (for the more complex plan)

```
-- Using the plan with outer rows=2:
SELECT 0.43 + 0.56 AS startup_cost,
     round((
         12.46 +           -- Outer scan cost
         2 * 16.57 +       -- Inner scan cost * 2 outer rows
          6 * current_setting('cpu_tuple_cost')::real
     )::numeric, 2) AS total_cost;

-- Result: total_cost = 45.66
```

Crucial Difference: Unlike the `Materialize` example, the full cost of the inner index scan (16.57) is paid for each of the 2 outer rows because the result is not cached.

NotebookLM

Validating Estimates with EXPLAIN ANALYZE

SQL Query in Fira Code:

```
EXPLAIN (analyze, timing off, summary off) SELECT *
FROM tickets t
JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
WHERE t.book_ref = '03A76D';
```

Annotated ANALYZE Output

```
                                QUERY PLAN
-----------------------------------------------------------------------
Nested Loop (cost=0.99..45.68 rows=6 width=136) (actual rows=8 loops=1)
  ->  Index Scan … on tickets t (actual rows=2 loops=1)
        Index Cond: (book_ref = '03A76D'::bpchar)
  ->  Index Scan ... on ticket_flights tf (actual rows=4 loops=2)
        Index Cond: (ticket_no = t.ticket_no)
```

(4) `2 loops * 4 rows/loop = 8` total rows. The math checks out.

(2) **This is the proof.** The inner Index Scan node was executed exactly **twice**—once for each row returned by the outer scan.

(1) The planner's estimate of 2 outer rows was correct.

(3) On average, each execution of the inner scan returned 4 rows.

NotebookLM

Next Optimization: The `Memoize` Node Caches by Parameter

The Problem

What if the outer set contains duplicate values for the join key? For example, multiple flights using the same aircraft code. In a simple parameterized join, the inner node would be executed repeatedly with the *exact same parameter*, fetching the *exact same results* every time. This is redundant work.

The Solution: The `Memoize` Node

PostgreSQL introduces the `Memoize` node, a specialized caching layer designed specifically for parameterized joins. It wraps the inner plan node and stores the results in a hash table, keyed by the parameter value(s) from the outer join.

Feature	`Materialize`	`Memoize`
Awareness	Parameter-unaware; caches the *entire* result set.	Parameter-aware; caches *multiple* result sets per unique key.
Mechanism	Simple in-memory buffer.	In-memory hash table, keyed by parameter value.
Memory	Spills to temporary disk file if `work_mem` is full.	Strictly in-memory; has an eviction policy to remove cold entries.
Use Case	Non-parameterized inner loops.	Parameterized inner loops with repeating key values.

NotebookLM

`Memoize` in the Query Plan

SQL Query in Source Serif Pro

```
EXPLAIN SELECT *
FROM flights f
JOIN aircrafts_data a ON f.aircraft_code = a.aircraft_code
WHERE f.flight_no = 'PG0003';
```

Annotated Query Plan in Source Serif Pro

```
                                QUERY PLAN
-----------------------------------------------------------------------
Nested Loop (cost=5.44..387.10 rows=113 width=135)
  ->  Bitmap Heap Scan on flights f (cost=5.30..382.22 rows=113 ...)
      ...
  ->  Memoize (cost=0.15..0.27 rows=1 width=72)
       Cache Key: f.aircraft_code
       Cache Mode: logical
       ->  Index Scan using aircrafts_pkey on aircrafts_data a
            Index Cond: (aircraft_code = f.aircraft_code)
```

The planner has wrapped the inner `Index Scan` with a `Memoize` caching node.

This explicitly tells us which parameter value from the outer loop will be used as the key for the cache's internal hash table.

Note: The cost shown for the `Memoize` node itself is a planner placeholder, not the true cost of a cache hit or miss. The real performance benefit is only revealed during execution via `EXPLAIN ANALYZE`.

NotebookLM

Measuring Cache Effectiveness with `ANALYZE`

```
QUERY PLAN
-----------------------------------------------------------------
Nested Loop (actual rows=113 loops=1)
  -> Bitmap Heap Scan on flights f (actual rows=113 loops=1)
     ...
  -> Memoize (actual rows=1 loops=113)
      Cache Key: f.aircraft_code
      Hits: 112  Misses: 1  Evictions: 0  Overflows: 0
      Memory Usage: 1kB
  -> Index Scan ... on aircrafts_data a (actual rows=1 loops=1)
```

Interpreting the Cache Statistics

loops=113: The `Memoize` node was invoked 113 times, once for each row from the outer `flights` scan.

Misses: 1: The very first time a unique `aircraft_code` was seen, the data was not in the cache. `Memoize` had to execute its child `Index Scan` to fetch it. This is the single cache miss.

Hits: 112: For the next 112 calls, the same `aircraft_code` was presented. The data was found directly in the cache, and the underlying `Index Scan` was **not executed**. This is the core of the optimization.

actual rows=1 loops=1 (on Index Scan): This proves the child node was only physically executed once for the single cache miss.

Evictions / Overflows: Zeros here indicate the cache (`work_mem`) was large enough. High numbers would signal that the cache is thrashing, which could degrade performance.

NotebookLM

Algorithm Variations: Outer and Non-Equi Joins

Left Outer Join

The Nested Loop algorithm can perform Left Outer Joins, appearing as `Nested Loop Left Join` in the plan.

- **Cardinality Estimation**
  The planner knows an outer join can never produce *fewer* rows than its outer input set. Therefore, the final row estimate is calculated as: `MAX(inner_join_estimate, outer_set_cardinality)`.
- **Limitations**
  The algorithm's nature is asymmetric: it always fully scans the outer set while selectively probing the inner. Because of this, it can be used for `LEFT JOIN`, but not for `RIGHT JOIN` or `FULL JOIN`, which would require the ability to probe in both directions.

Non-Equi Joins

Because it is fundamentally a brute-force comparison method, the Nested Loop is the **only** algorithm that can handle arbitrary join conditions that go beyond simple equality.

- **Use Cases**
  This makes it essential for joins using operators like `>`, `<>`, `LIKE`, or complex geometric operators.
- **Example**
  Finding all airports within 100km of each other might use a join condition like: `... ON a1.coordinates <@> a2.coordinates < 100`
- **Typical Plan**
  This often results in a `Nested Loop` with a `Join Filter` and a `Materialize` node on the inner side to avoid re-scanning for every comparison.

NotebookLM

Final Variations: Efficient `EXISTS` and `NOT EXISTS`

The Core Optimization: Short-Circuiting

For Semi and Anti joins, we don't need to find *all* matches on the inner side; we only need to know if *at least one* exists. The Nested Loop can stop searching the inner set as soon as the outcome is certain.

Join Types

- **Anti Join (`NOT EXISTS`):** Appears as **`Nested Loop Anti Join`**. For an outer row, the inner loop scans for a match. The moment the *first match is found*, the scan stops immediately, and the outer row is discarded.
- **Semi Join (`EXISTS`):** Appears as **`Nested Loop Semi Join`**. For an outer row, the inner loop scans for a match. The moment the *first match is found*, the scan stops immediately, and the outer row is kept.

Proof with `EXPLAIN ANALYZE`

The `ANALYZE` output is presented below, demonstrating the optimization in action.

```
QUERY PLAN
-----------------------------------------------------------
 Nested Loop Semi Join (actual rows=9 loops=1)
   -> Seq Scan on aircrafts_data ml (actual rows=9 loops=1)
   -> Index Only Scan using seats_pkey on seats s
        (actual rows=1 loops=9)
```

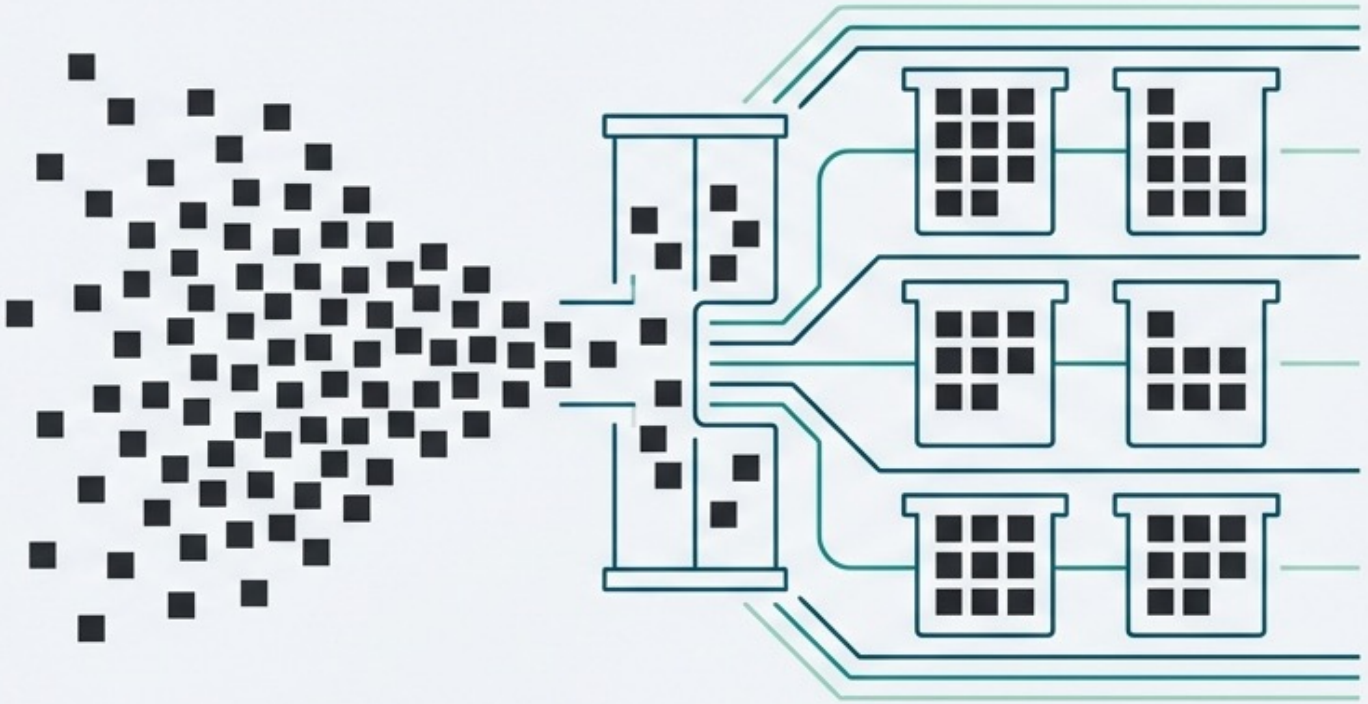
Key Insight: The inner `Index Only Scan` was executed 9 times (one for each aircraft), but each execution stopped after finding just **one row**, proving the short-circuiting behavior. This is what makes `EXISTS EXISTS` predicates far more efficient than a comparable `IN (SELECT ...)` or a standard `JOIN` when you only need to check for existence.

NotebookLM

22. Hashing

The Powerhouse of Database Joins: A Deep Dive into Hashing

Understanding the Mechanics of Hash Joins for Peak Query Performance



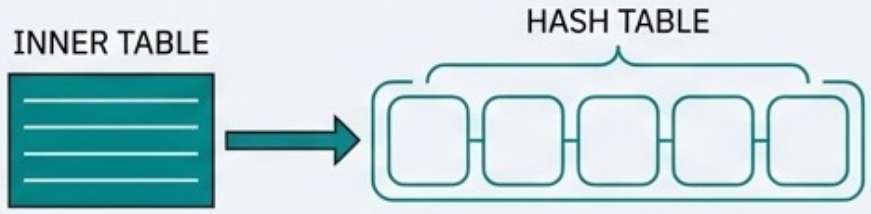
NotebookLM

The Two Acts of a Hash Join: Build & Probe

A hash join uses a pre-built hash table to find matching rows. The process is always broken into two distinct phases:

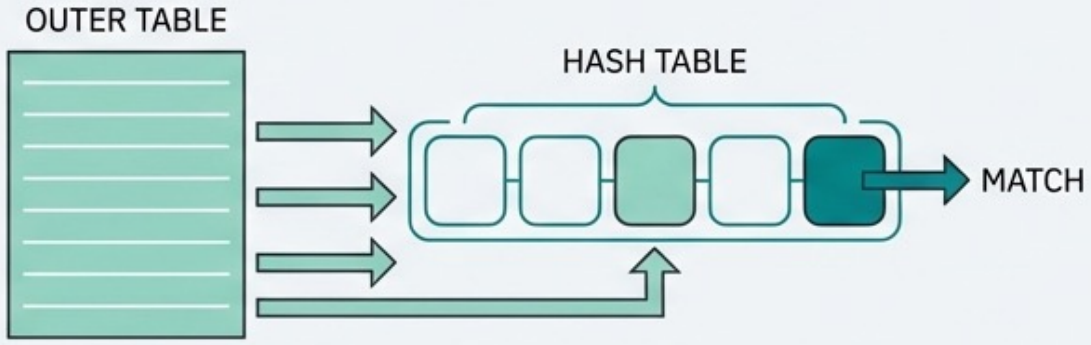
Act I: The Build Phase

- The database scans the **smaller (inner) table**.
- It uses the join key to build an in-memory hash table. The planner chooses the smaller table as the inner set to minimize the size of this hash table.



Act II: The Probe Phase

- The database scans the **larger (outer) table**.
- For each row, it calculates a hash of the join key and “probes” the hash table to find matches.
- Matching rows are returned to the parent node.



Key Characteristic

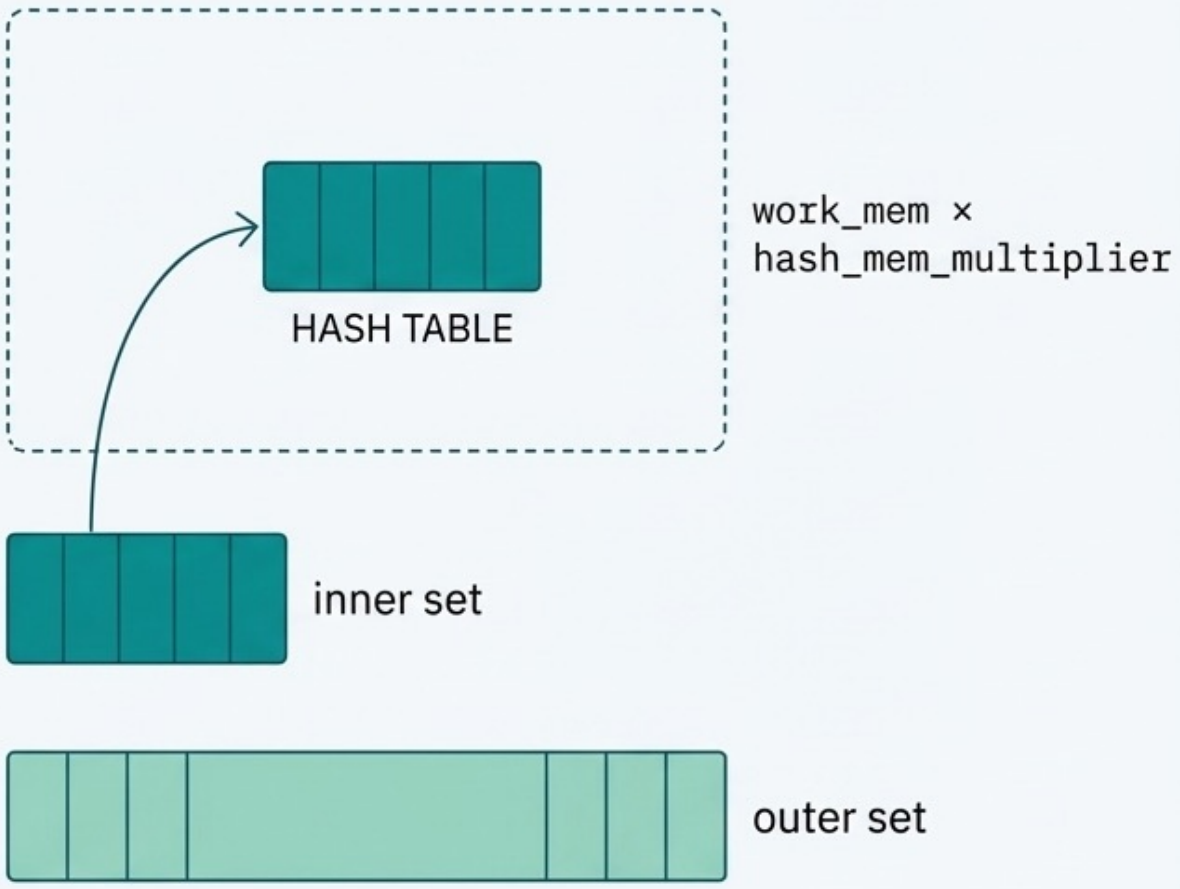
A hash join is a **"blocking" operator**. No rows can be returned until the Build phase is 100% complete.

NotebookLM

The 1-Pass Hash Join: When Everything Fits in Memory

The most efficient hash join occurs when the entire hash table (built from the inner set) fits completely within the allocated RAM. This is called a “1-Pass” or “in-memory” hash join.

- The amount of available memory for this operation is controlled by the `work_mem` parameter (default: 4MB), multiplied by `hash_mem_multiplier` (default: 1.0).
- The entire join is completed in a single pass (or ‘batch’).
- This is the fastest execution path for a hash join.



NotebookLM

Evidence in the Plan: Identifying a 1-Pass Join

The `EXPLAIN ANALYZE` output provides clear indicators of a 1-Pass join. Look for `Batches: 1`.

Example 1: `SELECT *`

```
EXPLAIN (analyze) SELECT * FROM bookings b JOIN tickets t ON b.book_ref = t.book_ref;

-> Hash (actual rows=2111110 loops=1)
     Buckets: 4194304  Batches: 1  Memory Usage: 145986kB
      -> Seq Scan on bookings b
```

The entire hash table required ~146MB and was processed in a single batch.

Memory footprint reduced by ~33MB.

Example 2: Selecting only necessary columns

```
EXPLAIN (analyze) SELECT b.book_ref FROM bookings b JOIN tickets t ON b.book_ref = t.book_ref;

-> Hash (actual rows=2111110 loops=1)
     Buckets: 4194304  Batches: 1  Memory Usage: 113172kB
      -> Seq Scan on bookings b
```

Key Takeaway: Selecting only the columns you need reduces the size of the hash table. This can be the difference between a fast 1-Pass join and a much slower 2-Pass join that spills to disk. Avoid SELECT * in performance-critical queries.

NotebookLM

Under the Hood: How the Planner Costs a 1-Pass Join

The planner's cost estimate for a hash join is a sum of specific CPU and I/O operations.

Startup Cost (cost=38.13..)

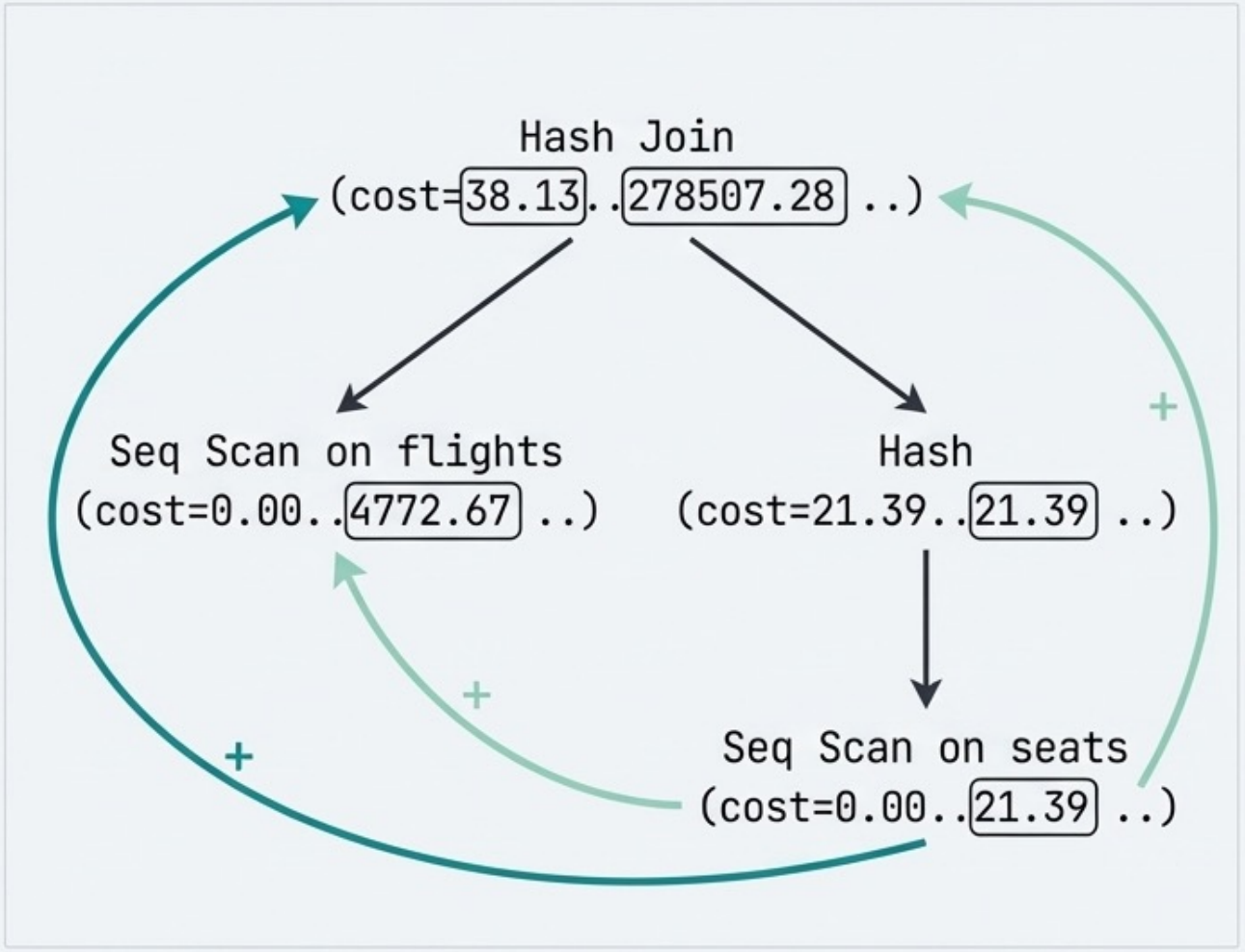
The cost to build the hash table.

1. **Cost to acquire inner set**: The total cost of the child scan (e.g., Seq Scan on seats cost is 21.39).
2. **Cost to hash inner rows**: cpu_operator_cost for each row in the inner set.
3. **Cost to insert into hash table**: cpu_tuple_cost for each row in the inner set.

Total Cost (..278507.28)

Startup Cost + The cost to probe and return results.

1. **Cost to acquire outer set**: The total cost of the outer scan.
2. **Cost to hash outer rows**: cpu_operator_cost for each row in the outer set.
3. **Cost to re-check join conditions**: Accounts for hash collisions.
4. **Cost to process result rows**: cpu_tuple_cost for each resulting row.



NotebookLM

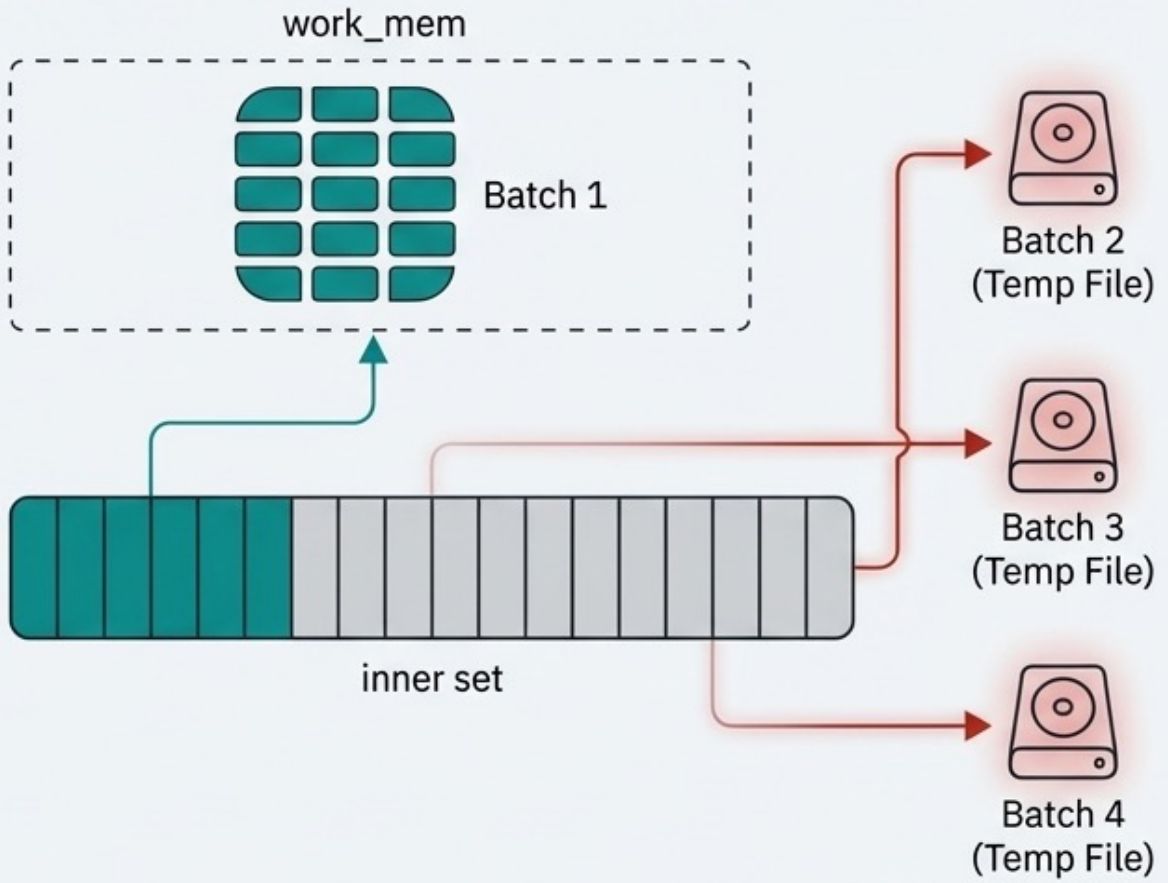
The 2-Pass Hash Join: When Memory Runs Out

Chapter 2: The Real-World Constraint

- When the planner estimates that the hash table will exceed work_mem, it switches to a 2-Pass algorithm. The goal is to break the large join into smaller chunks that can fit in memory one at a time.

The Process: Partitioning with Batches

1. The inner and outer sets are partitioned into multiple smaller **batches** using a hash function. The number of batches is always a power of two.
2. Rows belonging to the same join key will always be placed in the same batch.
3. For the **inner set**, Batch 1 is used to build a hash table in memory. All other batches (2, 3, 4...) are written to separate temporary files on disk. This is called **spilling**.



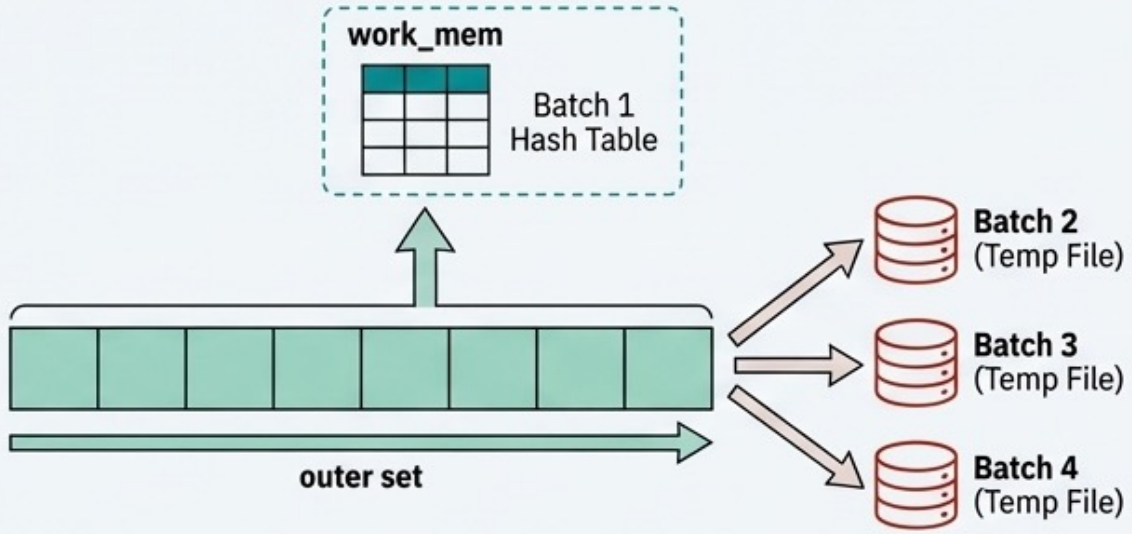
NotebookLM

The 2-Pass Mechanism: Probing and Processing Batches

After partitioning the inner set, the outer set is scanned.

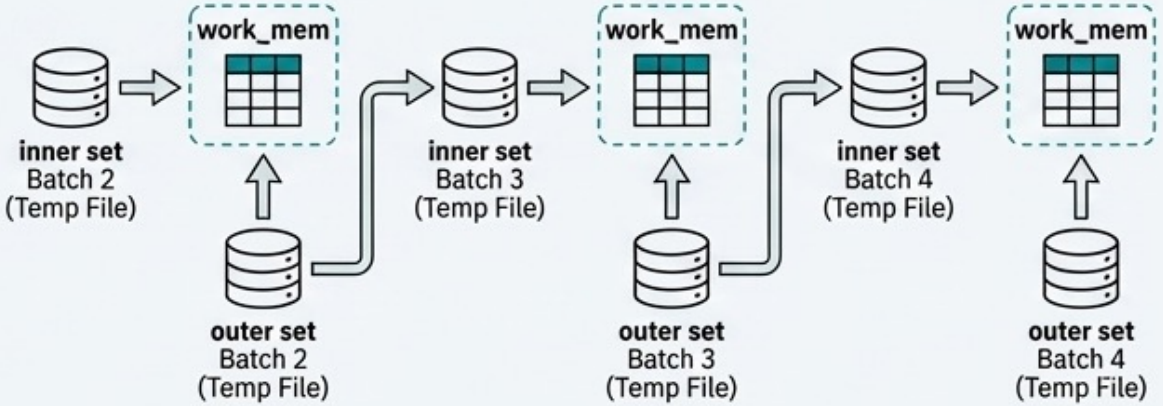
The Process: Probing and Spilling the Outer Set

1. As the **outer set** is scanned, rows in Batch 1 are immediately probed against the in-memory hash table.
2. Rows from all other batches (2, 3, 4...) are written to their own corresponding temporary files on disk.
3. After the outer scan is complete, the join for Batch 1 is done. The memory is cleared.



The Process: Looping Through Spilled Batches

4. The algorithm then iterates through the remaining batches. For each batch (e.g., Batch 2):
   - The inner set's temporary file is read to build a new hash table in memory.
   - The outer set's corresponding temporary file is read and probed against this new hash table.
   - Memory and temporary files are cleaned up.
5. This process repeats until all batches have been processed.



NotebookLM

Evidence in the Plan: Identifying a 2-Pass Join

A 2-Pass join leaves clear evidence in the query plan: multiple batches and temporary file I/O.

Query Plan Snippet (with default `work_mem` = 4MB)

```
Hash Join (actual rows=2949857 loops=1)
  Hash Cond: (t.book_ref = b.book_ref)
  Buffers: shared hit=7236 read=55626, temp read=55126 written=55126
  -> Seq Scan on tickets t ...
  -> Hash (actual rows=2111110 loops=1)
       Buckets: 65536  Batches: 64  Memory Usage: 2277kB
       Buffers: shared hit=7236 read=6211, temp written=10858
       -> Seq Scan on bookings b ...
```

Outer set and subsequent batches read/written to disk.

Data was split into 64 chunks.

Inner set spilled to disk during build phase.

****Key Indicators to Look For**:**

- `Batches: > 1`
- `temp written` in the Hash node.
- `temp read` and `temp written` in the `Hash Join` node.

This I/O is expensive and is a primary target for query tuning.

NotebookLM

Dynamic Adjustment: When Planner Estimates Are Wrong

The planner can be wrong due to stale or inaccurate statistics. If a hash table grows larger than expected and exceeds `work_mem`, the executor can adapt dynamically.

The Mechanism: On-the-Fly Batch Increase

- If the in-memory hash table for a batch grows too large, the executor will double the number of batches mid-execution.
- The current batch is split in two: roughly half the rows remain in the hash table, and the other half are moved to new temporary files.
- This can happen even if a 1-Pass join was originally planned.

Example: Underestimated Row Count

The planner thought 8 batches would be enough, but during execution, it had to increase to 32.

```
-> Hash (actual rows=2111110 loops=1)
      Buckets: 65536 (originally 65536)
      Batches: 32 (originally 8)
      Memory Usage: 4040kB
      -> Seq Scan on bookings_copy b ...
```

Planner's initial estimate was wrong. Executor adapted by quadrupling the batch count during execution.

Key Insight: While the database is resilient, this dynamic re-batching causes performance overhead. Accurate table statistics are crucial for the planner to choose the right number of batches from the start.

NotebookLM

Standard Parallel Join: Shared Nothing, Replicated Work

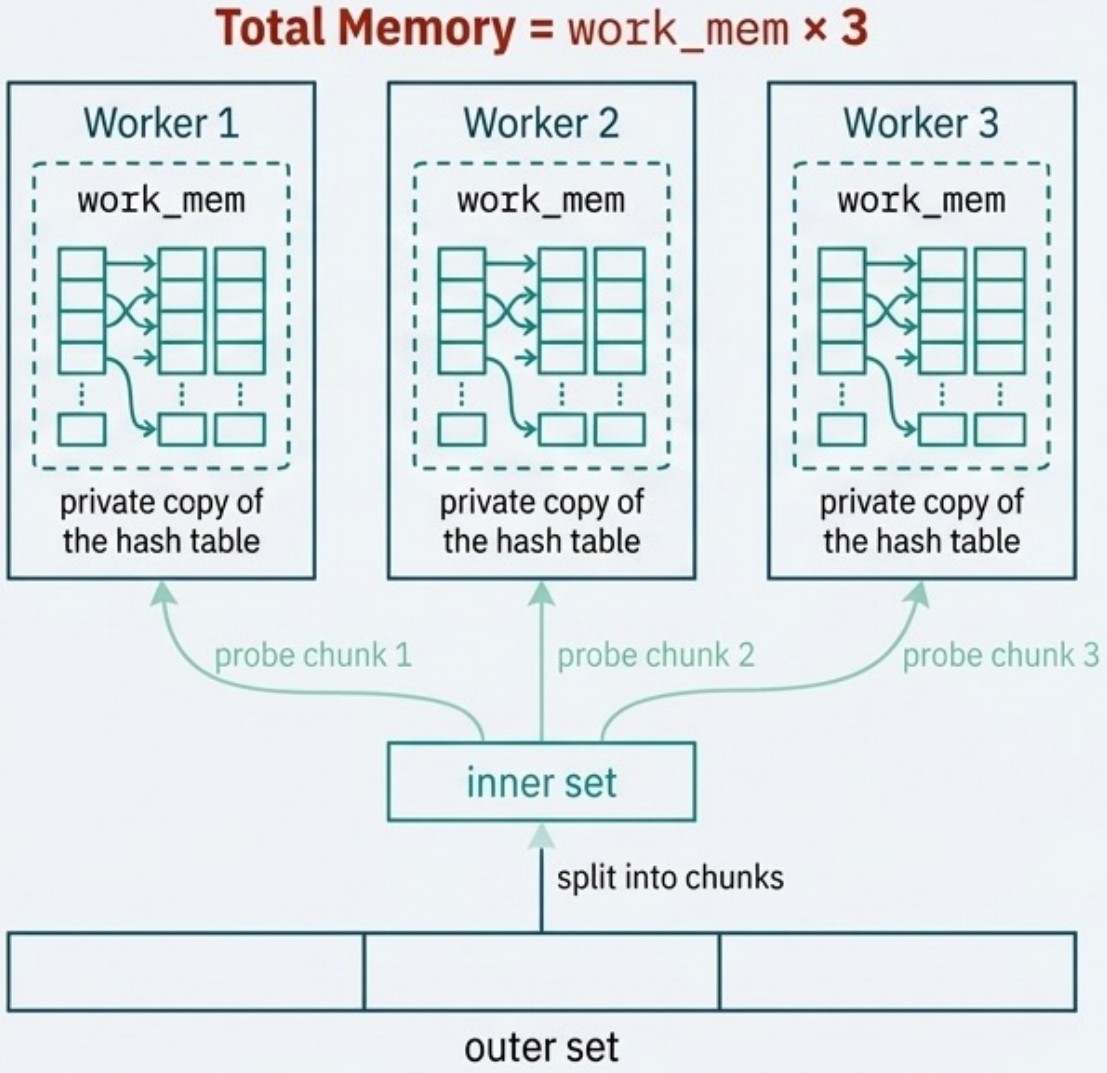
To use multiple CPU cores, the hash join algorithm can be parallelized. The most basic strategy involves multiple processes working independently.

The Mechanism

1. **Replicated Build**: Each parallel worker process independently scans the *entire* inner table and builds its own identical, private copy of the hash table in its own memory.
2. **Partitioned Probe**: The outer table is split, and each worker scans its assigned portion, probing it against its local hash table.

The Trade-off

- **Pro**: The expensive probe phase is parallelized effectively.
- **Con**: The build work is duplicated across all workers, which is inefficient for large inner tables. The total memory consumed is `work_mem` × `number of workers`.



NotebookLM

Parallel Hash Join: Shared Everything, Collaborative Work

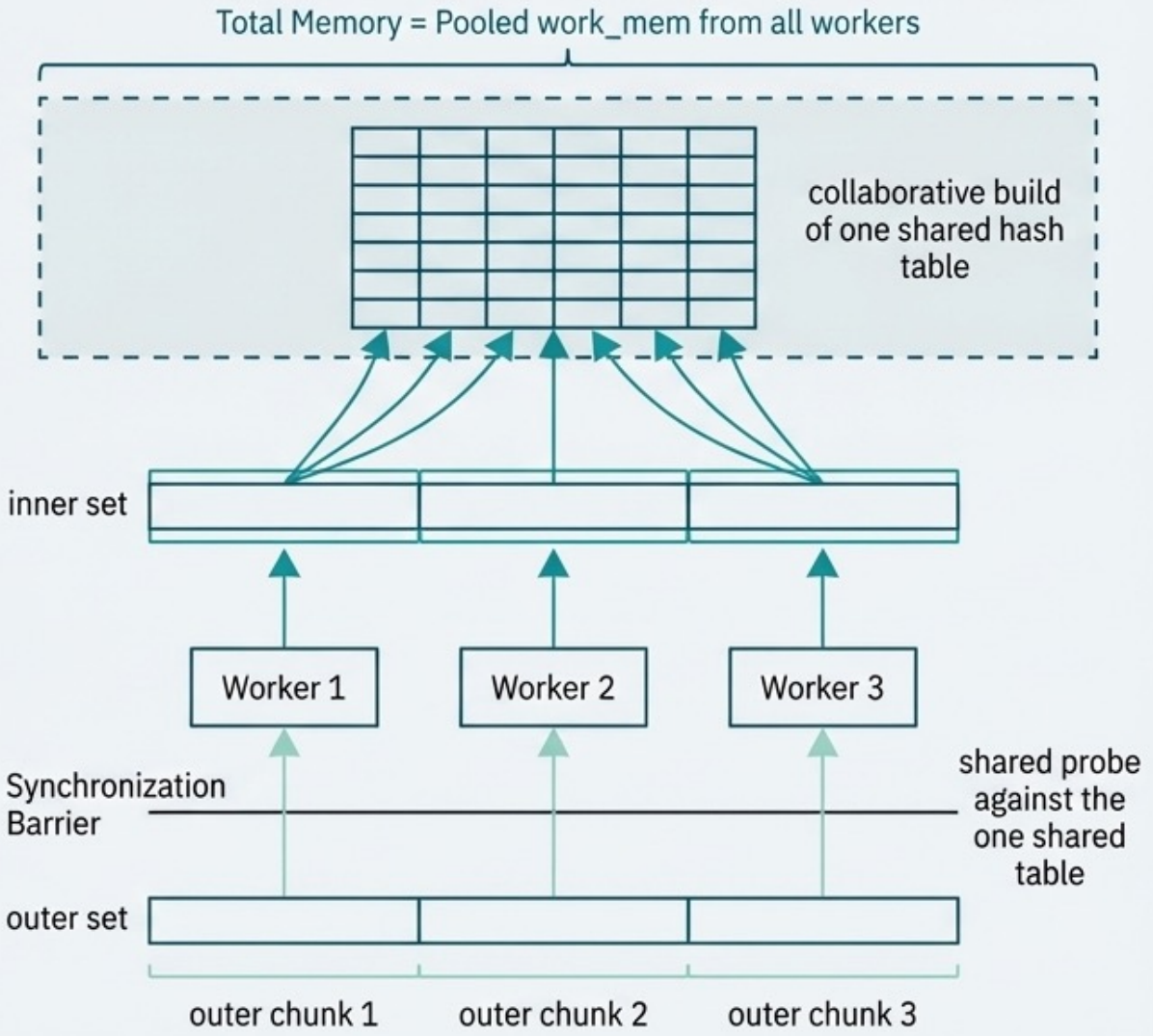
A more advanced algorithm (Parallel Hash) avoids redundant work by using shared memory. This is the default parallel strategy when **enable_parallel_hash** is on.

The Mechanism

1. **Collaborative Build**: All parallel workers cooperate to build a single, common hash table in a shared memory segment. They perform a parallel scan of the inner table together.
2. **Shared Probe**: After a synchronization barrier ensures the shared hash table is fully built, all workers perform a parallel scan of the outer table, probing their assigned rows against the one shared table.

The Advantage

- The build phase is also parallelized, eliminating redundant work.
- Memory from all workers is effectively pooled, dramatically increasing the size of the hash table that can fit in memory and making a 1-Pass join more likely.



NotebookLM

Evidence in the Plan: Comparing Parallel Strategies

The query plan clearly shows the difference. Even with half the `work_mem`, the collaborative `Parallel Hash` can achieve a 1-Pass join where the standard approach might spill.

Case 1: Standard Parallel Join

```
enable_parallel_hash = off
work_mem = 128MB
```

Each of the 3 workers builds its own 113MB hash table. Total memory usage: ~339MB.

```
-> Hash Join (actual rows=983286 loops=3)
    -> Hash (actual rows=2111110 loops=3)
         Buckets: 4194304  Batches: 1
         Memory Usage: 113172kB
```

3 × 113MB ≈ 339MB

Case 2: Specialized Parallel Hash Join

```
enable_parallel_hash = on
work_mem = 64MB
```

All 3 workers collaborate on a single 115MB hash table in shared memory. Total memory usage: ~115MB.

```
-> Parallel Hash Join (actual rows=983286 loops=3)
    -> Parallel Hash (actual rows=703703 loops=3)
         Buckets: 4194304  Batches: 1
         Memory Usage: 115392kB
```

Total usage is what you see.

Key Insight
The specialized Parallel Hash Join is more memory-efficient and scalable, making it more likely to perform a fast, 1-Pass join on large datasets.

NotebookLM

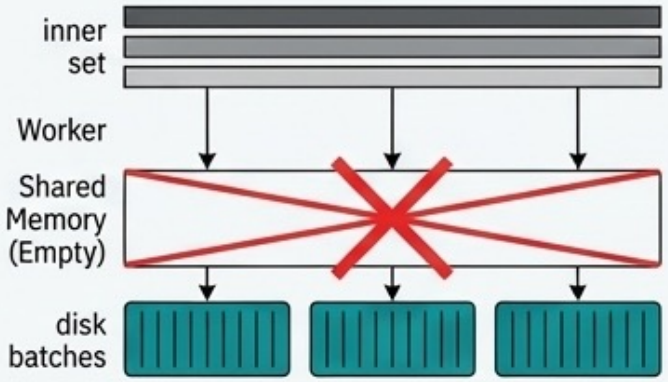
The Ultimate Challenge: 2-Pass Parallel Hash Join

When the combined memory of all workers is still not enough, a specialized 2-Pass Parallel Hash algorithm is used. Its mechanics are very different from the non-parallel version.

The Mechanism

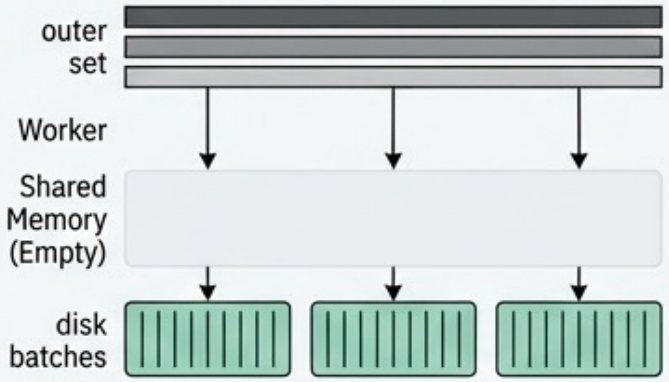
1. Parallel Partitioning (Inner Set)

Workers perform a parallel scan of the inner set, partitioning all rows into batches and writing them *directly* to shared temporary files on disk. **No hash table is built in memory during this phase, not even for Batch 1.**



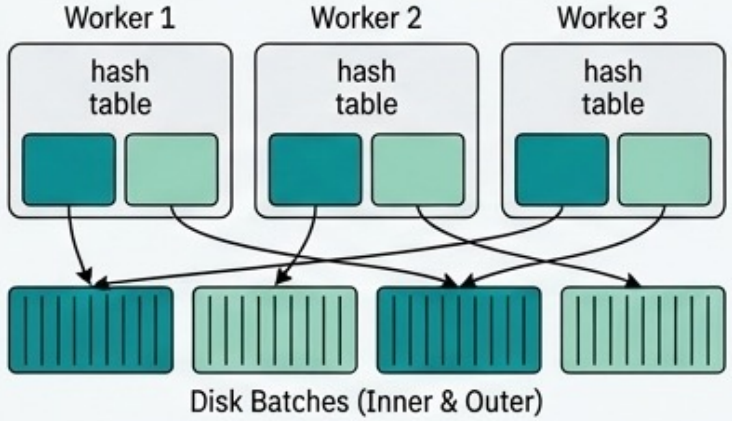
2. Parallel Partitioning (Outer Set)

After the inner set is fully partitioned to disk, workers do the same for the outer set, writing its rows to corresponding batch files.



3. Parallel Batch Processing

With all data staged on disk in batches, workers now pick up batches one by one. For each batch: a worker builds an in-memory hash table from the inner batch file and probes it with the outer batch file. Workers can collaboratively process batches.



****Key Difference****

Unlike the single-threaded 2-pass join, this method writes *all* batches to disk first to maximize parallel processing during the partitioning phase before any joining occurs.

NotebookLM

Beyond Inner Joins: Hashing's Other Roles

The hashing mechanism is not limited to inner joins. It's a fundamental building block for many database operations.

1. Outer Joins

Hash joins support `LEFT`, `RIGHT`, and `FULL OUTER` joins. The planner will transform the join type to ensure the smaller table is used for the build phase (e.g., a logical `LEFT JOIN` may be executed as a physical `Hash Right Join`).

```
-- A logical LEFT JOIN...
EXPLAIN (costs off) SELECT * FROM bookings
LEFT OUTER JOIN tickets ...

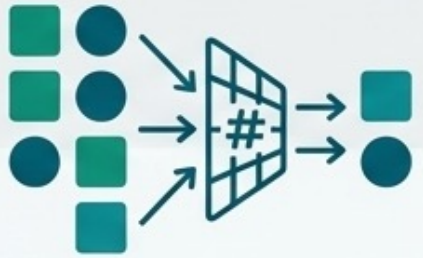
-- ...becomes a physical RIGHT JOIN to build on
-- the smaller 'bookings' table
QUERY PLAN
----------------
Hash Right Join
  Hash Cond: (t.book_ref = b.book_ref)
  -> Seq Scan on tickets t
  -> Hash
     -> Seq Scan on bookings b
```

2. Aggregation & Uniqueness

The `HashAggregate` operator uses a hash table to perform `GROUP BY`, `DISTINCT`, and `UNION` operations by storing only unique values. It uses the same `work_mem` limits and can also spill to disk in batches if memory is exceeded.

```
EXPLAIN (costs off) SELECT DISTINCT fare_conditions
FROM seats;

QUERY PLAN
----------------
HashAggregate
  Group Key: fare_conditions
  -> Seq Scan on seats
```



NotebookLM

Key Takeaways for Hash Join Performance Tuning

Understanding the mechanics of hash joins translates directly into writing and tuning better queries.



1. Control Your Memory Footprint.

- The goal is always a **1-Pass join**. The most direct way to achieve this is to ensure the hash table fits in `work_mem`.
- **Increase** `work_mem` for sessions running large analytical queries.
- **SELECT only the columns you need.** Avoid `SELECT *` to reduce the size of the rows stored in the hash table.



2. Ensure Accurate Planner Estimates.

- The planner relies on table statistics to choose the right join method and estimate the hash table size.
- **Keep statistics up-to-date** with `ANALYZE`. Inaccurate stats can lead to a 1-pass plan that dynamically re-batches, or a sub-optimal 2-pass plan.



3. Let the Planner Choose the Build Table.

- The planner will **automatically choose the** smaller of the two tables as the inner (build) set to minimize hash table size. Trust it to make this decision.

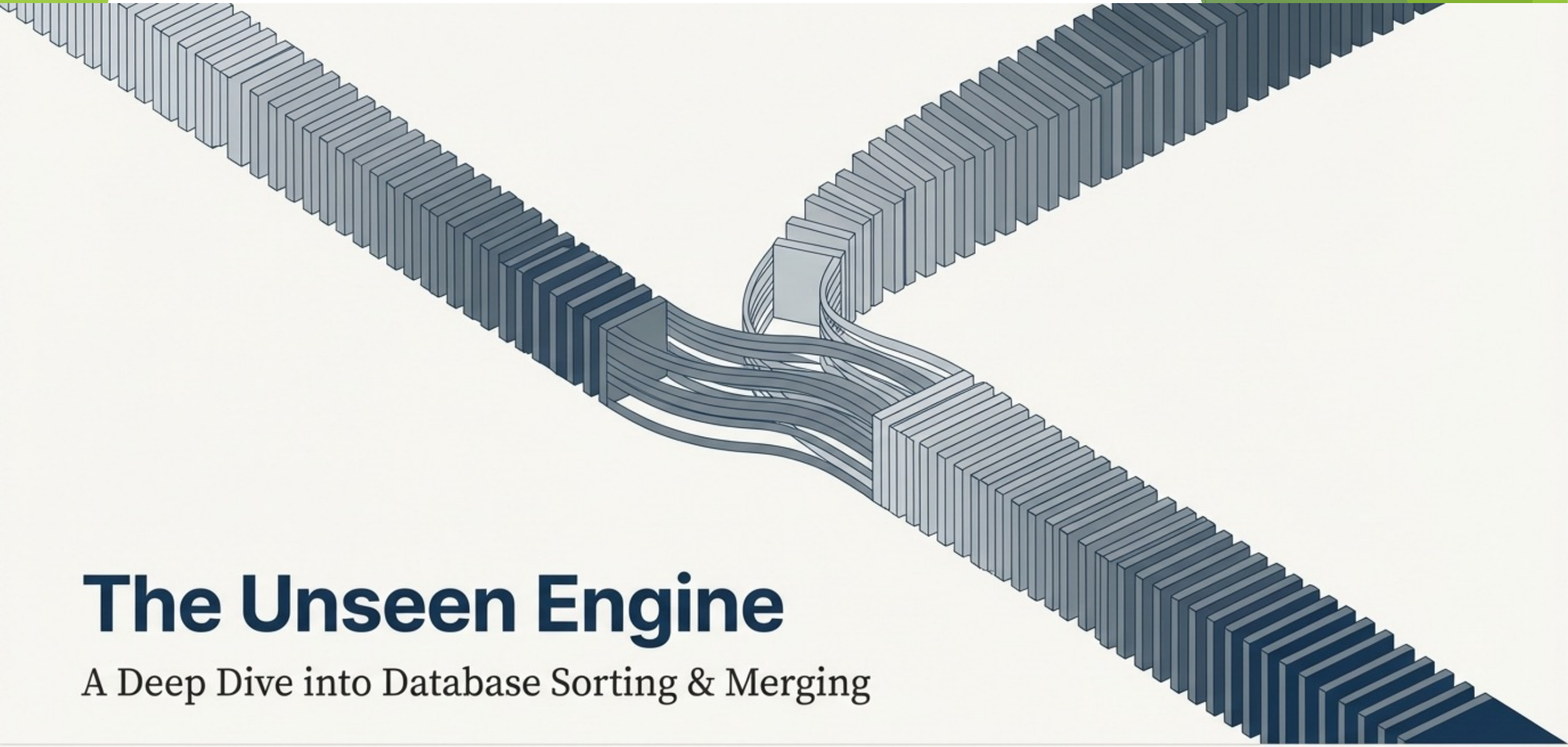


4. Leverage Modern Parallelism.

- The specialized **Parallel Hash Join** is highly efficient, pooling memory memory to make 1-Pass joins more likely. Ensure it is enabled (`enable_parallel_hash = on`).

NotebookLM

23. Sorting and Merging



The Unseen Engine

A Deep Dive into Database Sorting & Merging

Behind every query is a world of complex algorithms. Let’s explore the ones that bring order to data.

NotebookLM

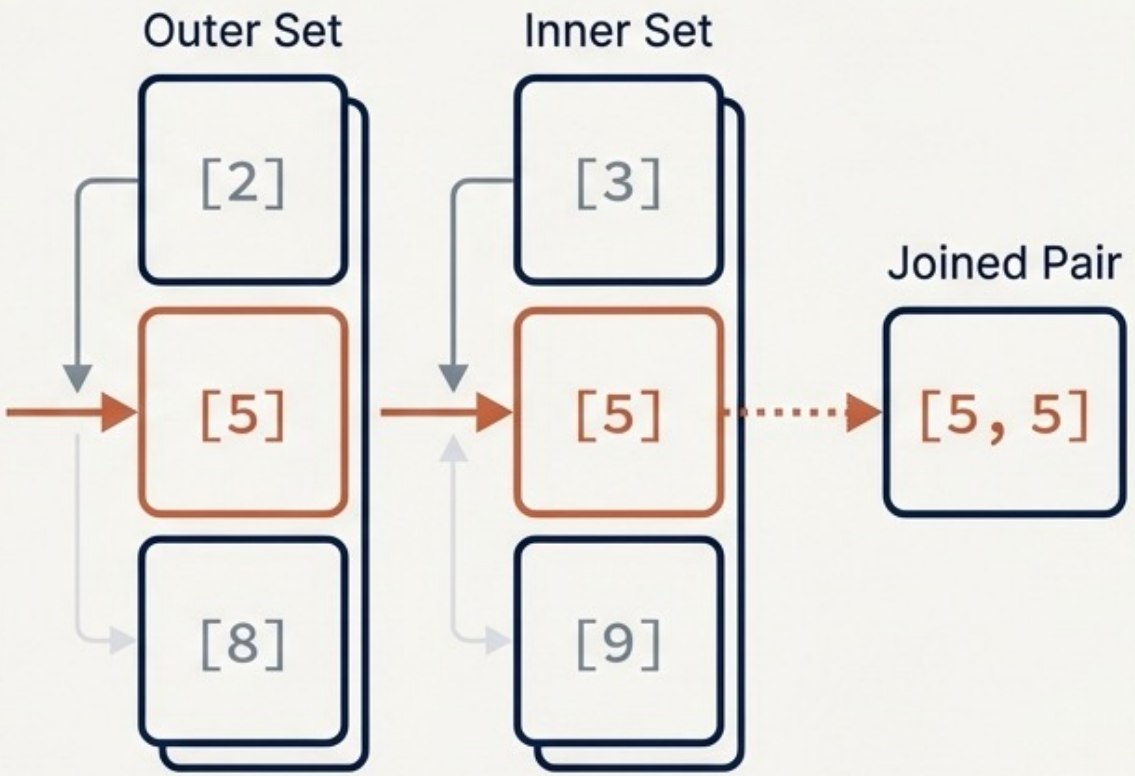
The Merge Join: An Elegant Dance of Sorted Data

A high-performance algorithm for joining two datasets that are **already sorted** on their join keys. It produces a result that is also sorted.

Mechanism:
It iterates through both sorted sets simultaneously, like merging two sorted decks of cards. This requires only a single pass over the data and uses no additional memory.

Key Requirements:

1. **Sorted Inputs:** Both datasets must be sorted on the join key. This can be achieved via an index scan or an explicit sort operation.
2. **Equality Condition:** The join condition must be an equality (an "equi-join").

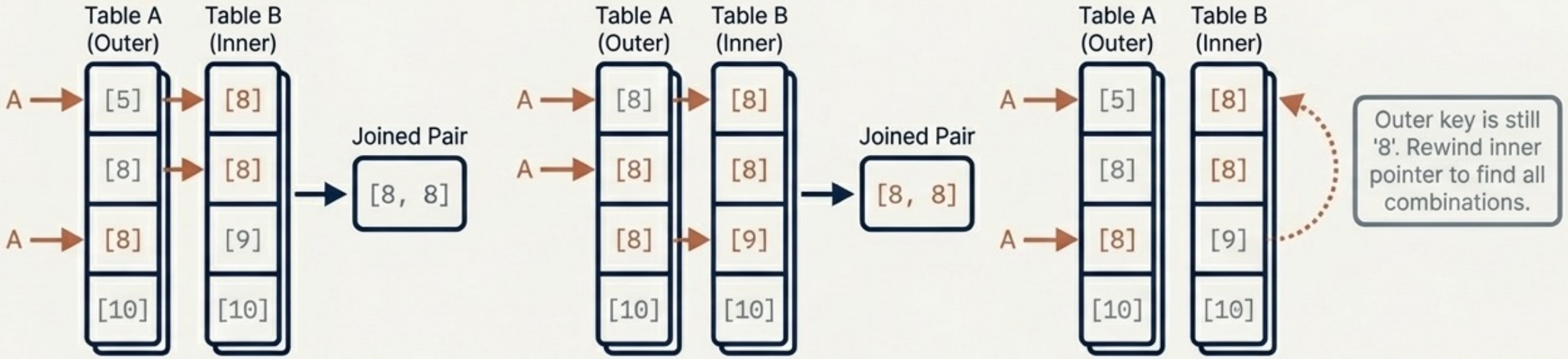


NotebookLM

One Pass, No Wasted Effort

The algorithm uses two "pointers," one for each dataset, to find matching rows in a single pass.

1. **Compare:** The values at the current position of each pointer are compared.
2. **Advance:** The pointer associated with the smaller value is advanced to the next row.
3. **Match:** If the values are equal, the joined row is returned, and the pointer for the inner dataset is advanced.
4. **Handle Duplicates:** This is the critical refinement. If the outer pointer advances but its key value *remains the same*, the inner pointer is "rewound" to the first matching row in its group. This ensures that every matching combination is found.



NotebookLM

Reading the Plan: The Optimizer's Smart Choice

The query optimizer prefers a Merge Join when inputs are naturally sorted by an index, as this avoids a costly explicit sort operation. This is especially true if the query also requests sorted output (`ORDER BY`), as the join naturally preserves that order.

```
-- QUERY: SELECT * FROM tickets t JOIN ticket_flights tf ON
tf.ticket_no = t.ticket_no ORDER BY t.ticket_no;

                              QUERY PLAN
-----------------------------------------------------------------
-> Merge Join
     Merge Cond: (t.ticket_no = tf.ticket_no)
    -> Index Scan using tickets_pkey on tickets t
    -> Index Scan using ticket_flights_pkey on ticket_flights tf
```

The chosen join strategy, selected for its efficiency with sorted inputs.

This is the key. The index provides a pre-sorted data stream, eliminating the need for a separate Sort step.

NotebookLM

What Happens When Data Isn't Sorted?

If a suitable index doesn't exist to provide sorted data, the planner has no choice but to sort it first. This preparatory work appears in the plan as an explicit `Sort` node. A Merge Join cannot begin until both of its input-side sort operations are faly complete, adding significant startup cost.

```
-- QUERY: SELECT * FROM flights f JOIN airports_data dep ON f.departure_airport = dep.airport_code
ORDER BY dep.airport_code;

                          QUERY PLAN
-------------------------------------------------------------
 -> Merge Join
      Merge Cond: (f.departure_airport = dep.airport_code)
      -> Sort
           Sort Key: f.departure_airport
           -> Seq Scan on flights f
      -> Sort
           Sort Key: dep.airport_code
           -> Seq Scan on airports_data dep
```

This is the performance **cost.** The efficiency of these sort operations is now critical to the entire query.

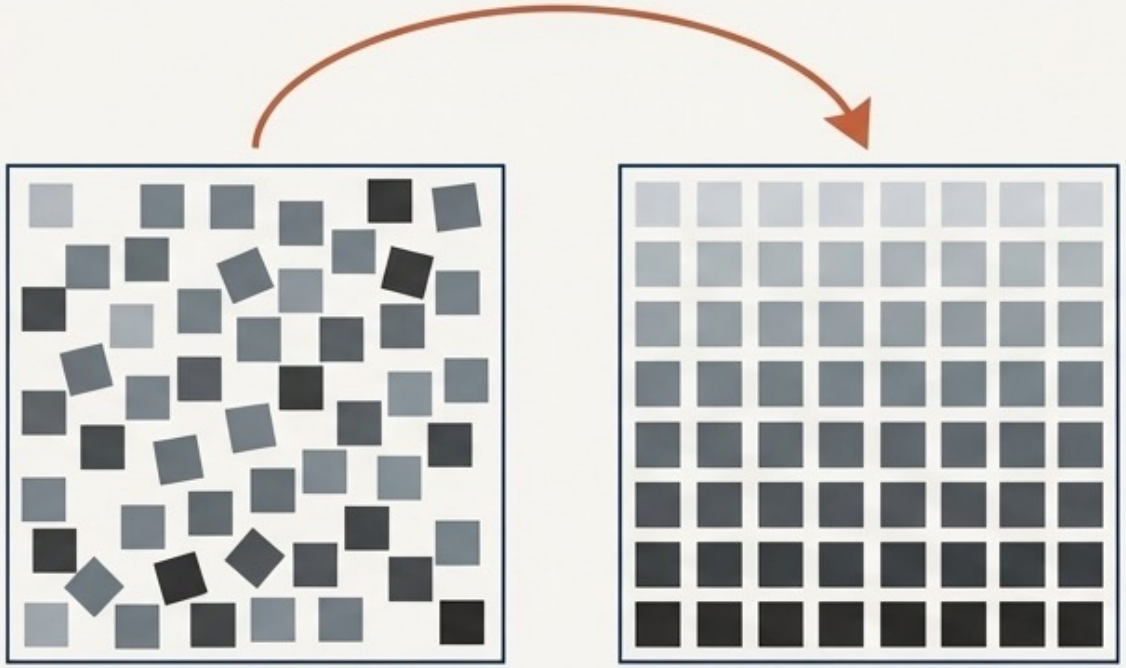
NotebookLM

The Sorting Toolkit: High-Speed In-Memory Operations

When data fits into the memory allocated by `work_mem`, the database uses highly optimized algorithms.

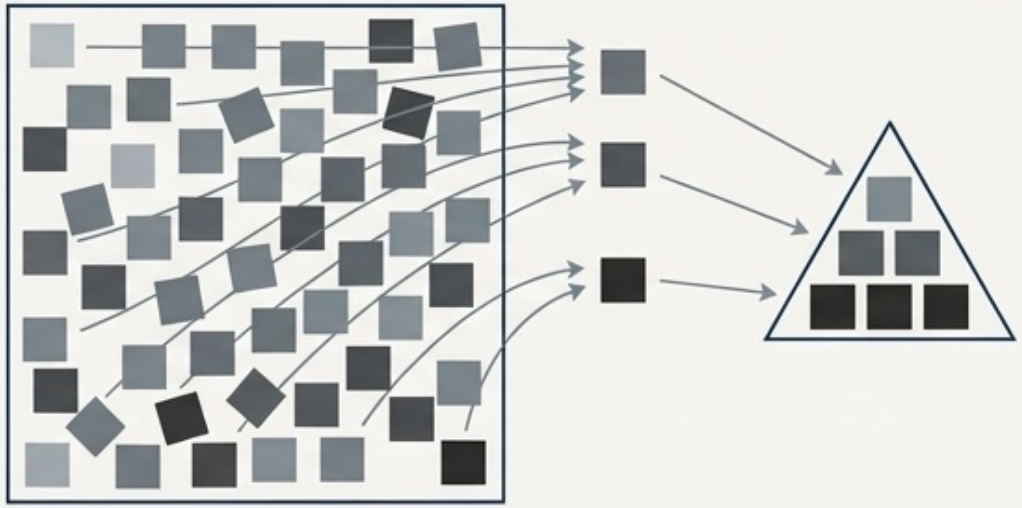
Quicksort (`quicksort`)

The default workhorse. Its average-case O(n log n) complexity is efficient for general-purpose sorting of a full dataset. Used when the entire dataset must be sorted.



Top-N Heapsort (`top-N heapsort`)

A specialist algorithm for queries with a `LIMIT` clause. Instead of sorting the entire dataset, it maintains a small binary heap of only the top k items. This reduces complexity to O(n log k), making it significantly faster when k is much smaller than n.

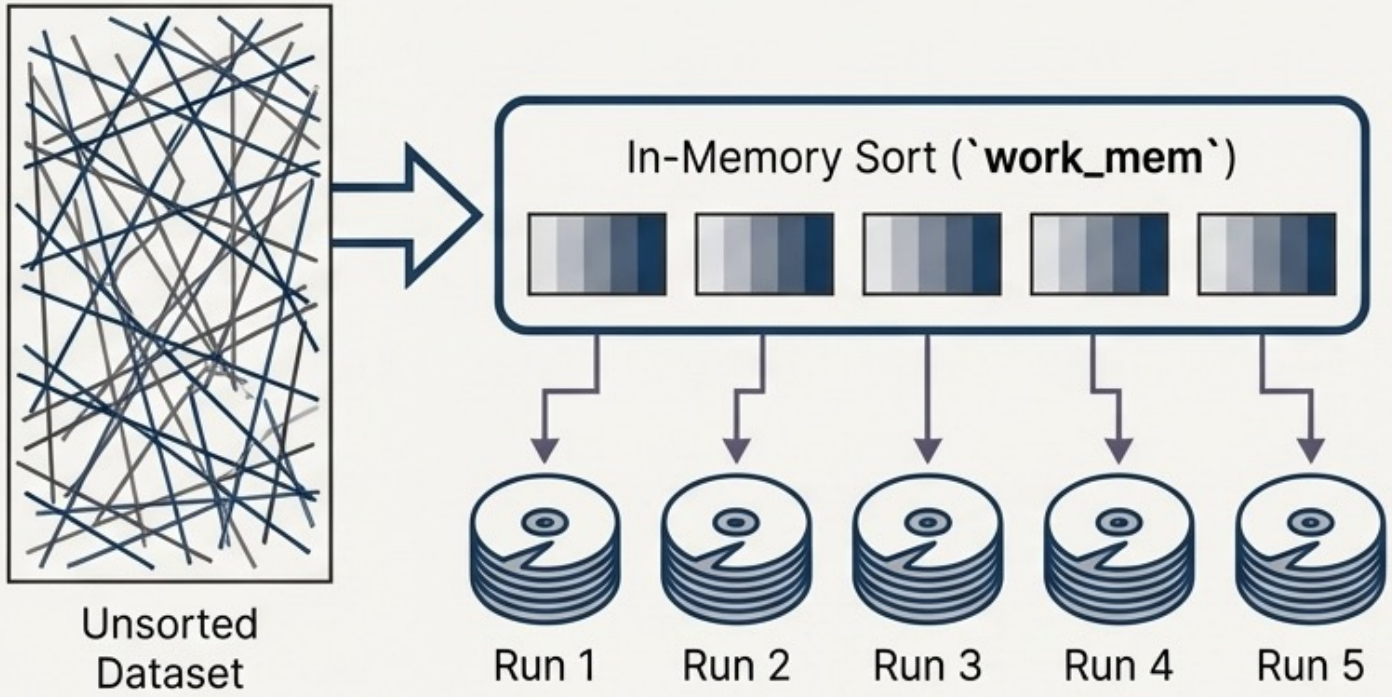


NotebookLM

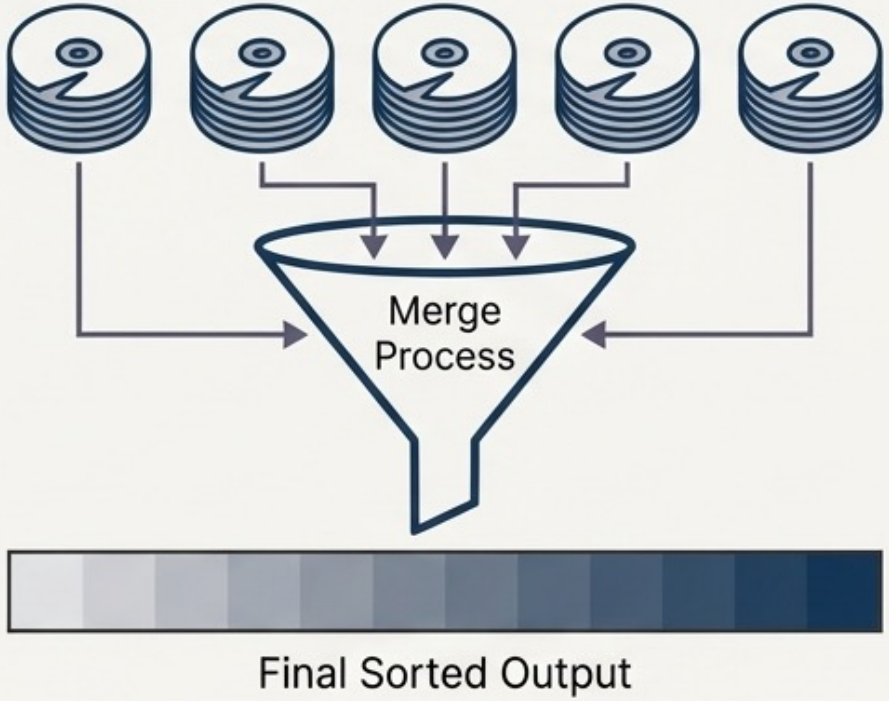
Sorting Datasets Larger Than Memory: External Merge Sort

When a dataset exceeds the `work_mem` limit, the database switches to a disk-based strategy called **External Merge Sort**.

1 Step 1: Create Runs



2 Step 2: Merge Runs



Key Insight: The amount of `work_mem` is crucial. More memory allows for larger initial runs, which means fewer runs overall. Fewer runs require fewer merge passes, dramatically reducing I/O and speeding up the sort.

NotebookLM

Don’t Re-Sort What’s Already Ordered: Incremental Sort

Use Case: This method is used when data is already sorted by the *first* key in a multi-key `ORDER BY` clause (e.g., `ORDER BY total_amount, book_date` where an index on `total_amount` exists).

Mechanism: It processes the data in groups based on the pre-sorted key (`total_amount`). Within each group of identical keys, it performs a much smaller and faster sort on the subsequent key (`book_date`).

Primary Benefits: Significantly lower memory usage compared to a full sort, and the ability to start returning rows as soon as the first group is processed (first-tuple-fast).

```
-- QUERY: SELECT * FROM bookings ORDER BY
total_amount, book_date;

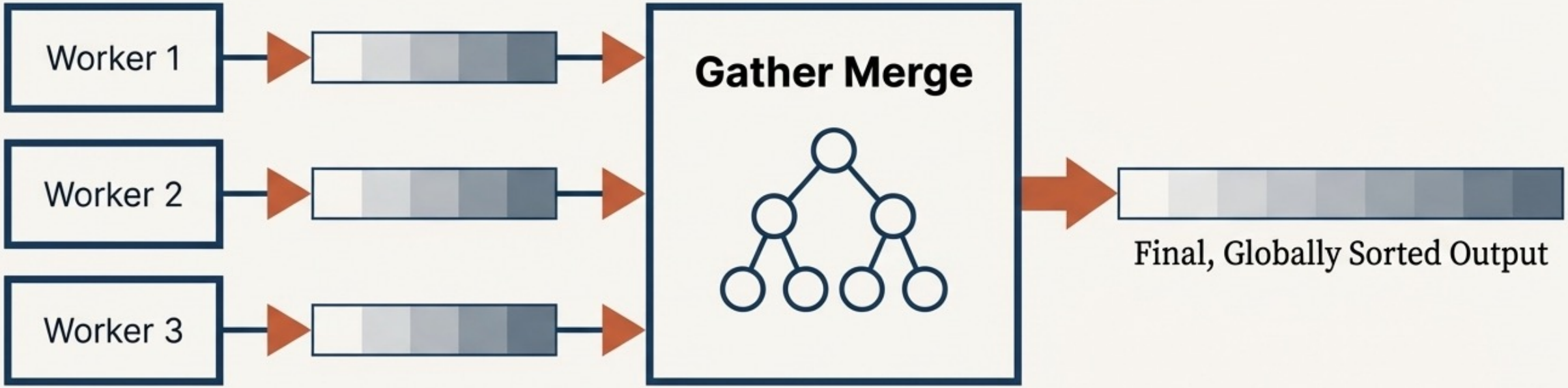
                    QUERY PLAN
-----------------------------------------
-> Incremental Sort
     Sort Key: total_amount, book_date
     Presorted Key: total_amount
      -> Index Scan using
bookings_total_amount_idx on bookings
-----------------------------------------
```

The planner recognizes the data is already ordered by this key from the index scan and uses it to avoid a full re-sort.

NotebookLM

Coordinating at Scale: Parallel Sorting and Merging

Sorting can be parallelized: multiple workers can sort different chunks of data simultaneously. However, the sorted results from these workers must be combined in the correct final order. A simple Gather node would interleave them incorrectly. The **Gather Merge** node solves this. It acts as a final merge point, efficiently merging the pre-sorted streams from each worker into a single, globally sorted output stream.

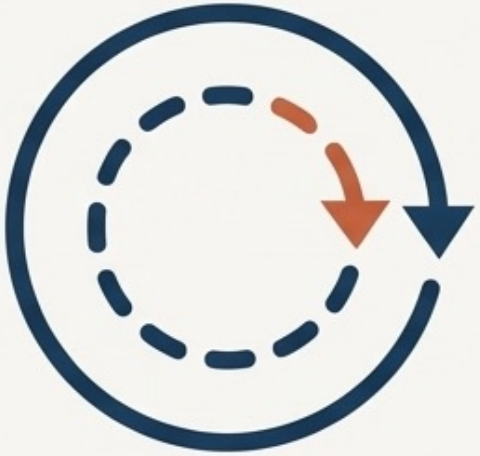


Under the hood: `Gather Merge' uses a binary heap to manage the next-available row from each worker stream, ensuring the global order is always maintained with minimal overhead.

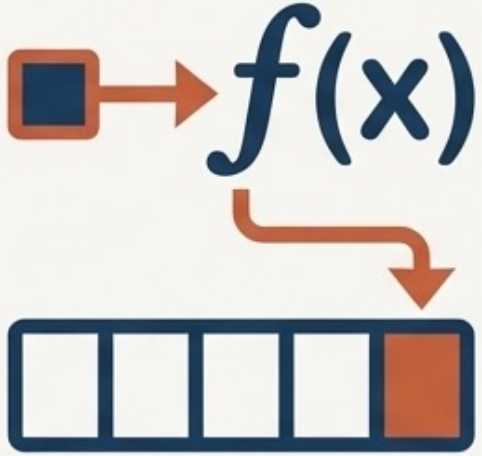
NotebookLM

Choosing the Right Tool: The Three Join Contenders

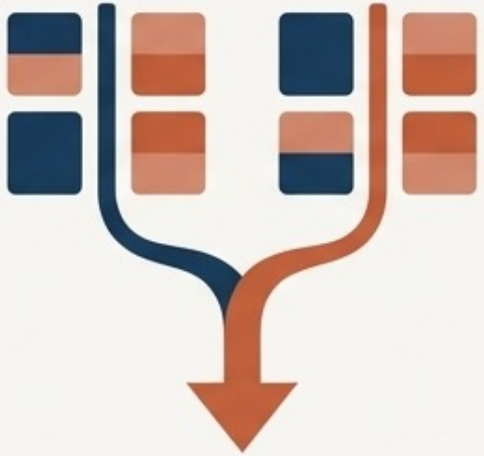
The query planner's primary task in a join is to choose between three distinct methods. Each has a unique operational model, leading to different strengths and weaknesses based on data size, structure, and available indexes.



Nested Loop



Hash Join



Merge Join

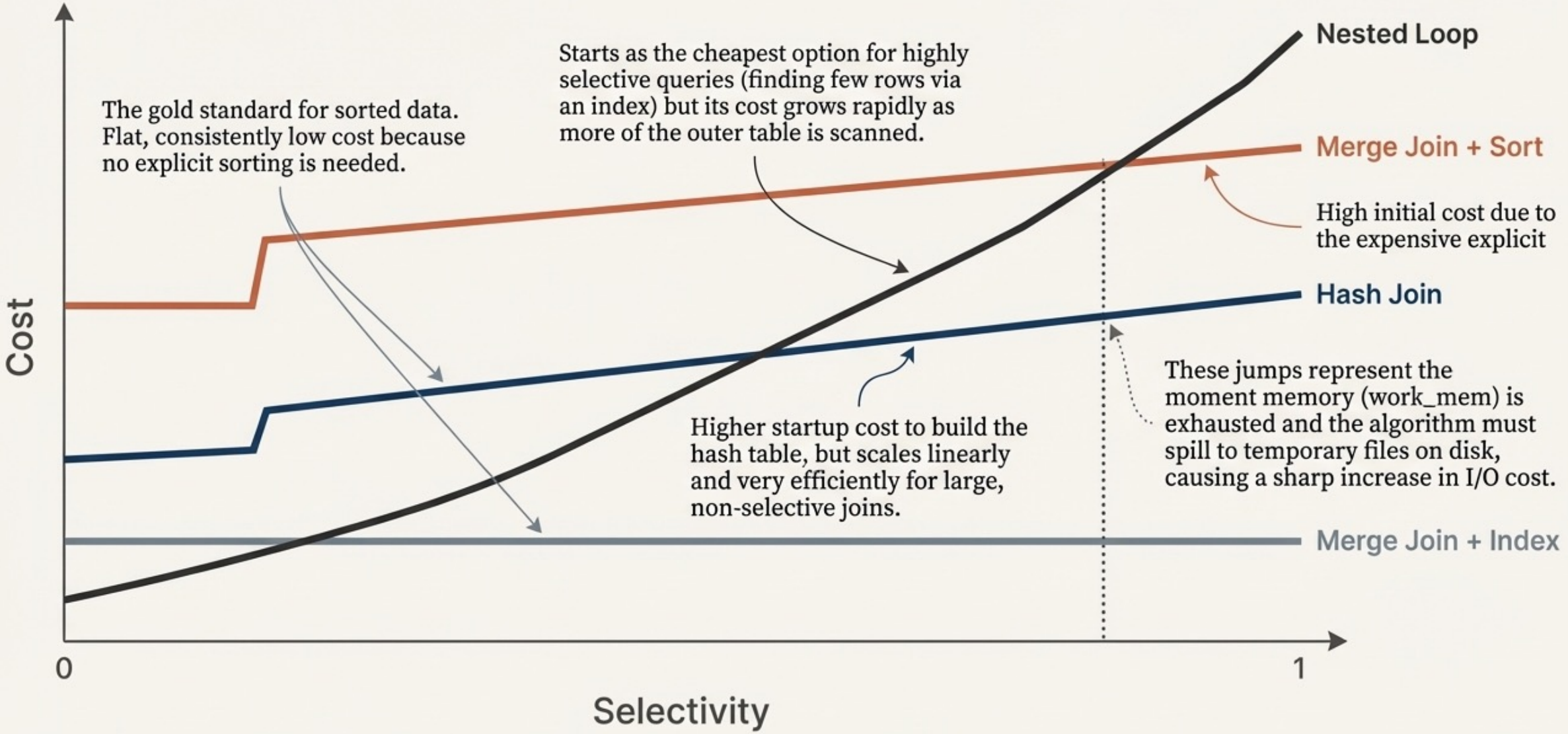
NotebookLM

Join Strategy Performance at a Glance

Method	Best For	Startup Cost	Memory Use	Key Requirement
 **Nested Loop**	Small outer sets, indexed inner sets (OLTP queries).	**Lowest.** Can return the first row almost instantly.	Low	None. Universal, supports all join conditions.
 **Hash Join**	Large, unsorted datasets (bulk OLAP queries).	**High.** Must build the entire hash table before returning rows.	High	Equi-join only. Join key must be hashable.
 **Merge Join**	Large, pre-sorted datasets.	**Low.** No pre-processing needed if inputs are sorted.	Low	Equi-join only. Inputs must be sorted.

NotebookLM

Visualizing the Trade-Offs: Cost vs. Selectivity



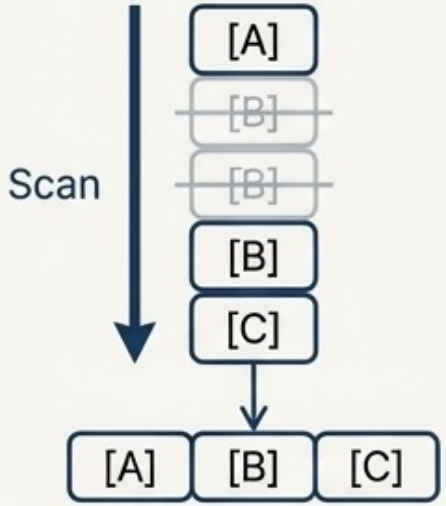
NotebookLM

Beyond Joins: Sorting as a Universal Grouping Tool

The principle of ordering data is a foundational strategy for many operations beyond joins. Once a dataset is sorted, grouping-based operations become trivial one-pass scans.

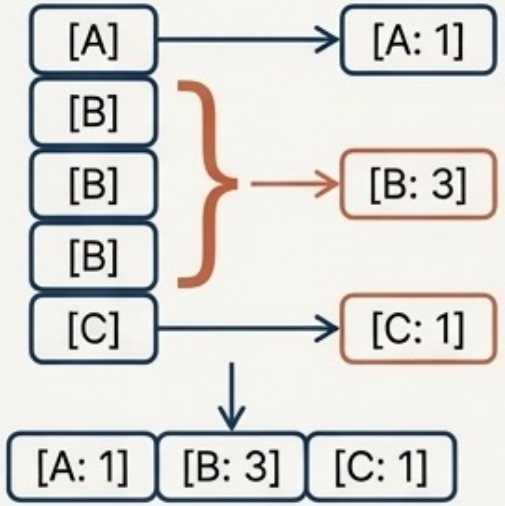
`DISTINCT`

Finding unique values in a sorted list is as simple as scanning and skipping any value that is the same as the previous one. The execution plan represents this with a `Unique` node.



`GROUP BY`

Similarly, once sorted by the grouping key, all rows for a given group are contiguous. This allows aggregation functions (like `COUNT(*)` or `SUM()`) to be calculated in a single pass. The plan shows this as a `GroupAggregate` node.



Efficient aggregation is deeply connected to data order. Hashing is one strategy; sorting is the other.

NotebookLM

Mastering the Flow of Data

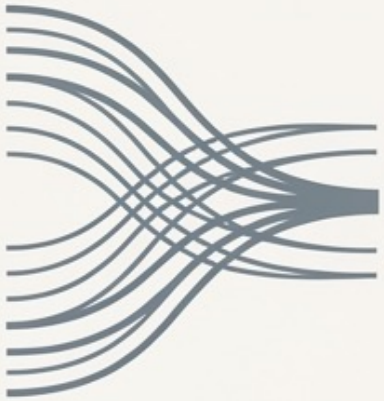
1. **Merge Join is elite for pre-sorted data.** It offers linear performance with minimal memory overhead, making it ideal when indexes can provide the necessary order.

2. **The `Sort node is a powerful, versatile engine.** It has a rich toolkit (`quicksort`, `top-N heapsort`, `external merge`, `incremental sort`) to prepare data for merging, aggregation, and ordering under any condition.

3. **The planner's choice is a complex trade-off.** The decision between **Nested Loop**, **Hash**, and **Merge Join** is a sophisticated balance of data size, selectivity, available indexes, `work_mem`, and required output order.

4. **`EXPLAIN` is your window into this world.** Use it to diagnose performance and understand *why* the database makes the choices it does.

NotebookLM



The Unseen Engine

“The best queries are written with an understanding of the work the database does on their behalf.”

NotebookLM

24. Hash



Anatomy of a PostgreSQL Hash Index

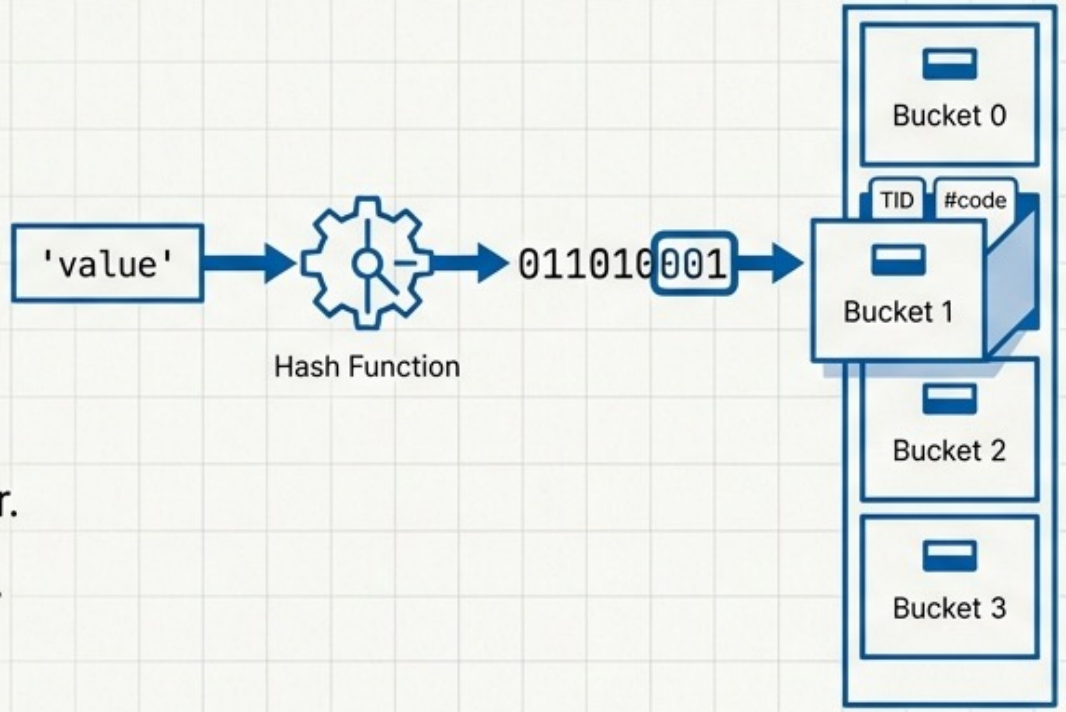
A Guided Tour of the High-Speed Directory

NotebookLM

Built for One Job: The Fastest Equality Search.

A Hash Index provides the quickest way to find a tuple ID (TID) for a specific key value. It is essentially a disk-based hash table.

Its sole supported operation is an equality search (`WHERE column = 'value'`).



How it works:

1. A hash function computes a code from the index key.
2. The lowest bits of this code determine a ‘bucket’ number.
3. The TID and the full hash code are stored in that bucket.

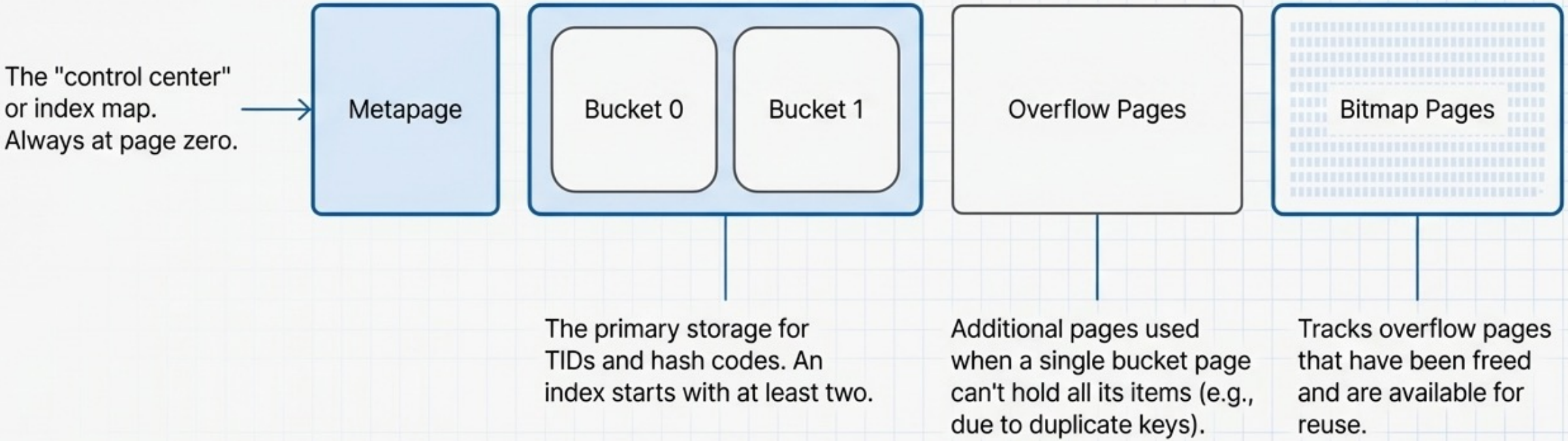
Key Detail:

The original key value is **not** stored in the index. This has important implications for index-only scans and re-checks.

NotebookLM

The Blueprint: Four Essential Page Types

A hash index organizes all its data into four distinct page types to minimize disk access during operations.



NotebookLM

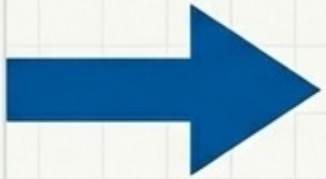
From `CREATE INDEX` to a Four-Page Reality.

Using the `pageinspect` extension, we can see the minimal structure of a newly created hash index on an empty, analyzed table.

The Code

```
-- Create the extension and table
CREATE EXTENSION pageinspect;
CREATE TABLE t(n integer);
ANALYZE t;

-- Create the index
CREATE INDEX t_n_idx ON t USING hash(n);
```



The Consequence

The index immediately contains four pages: one for each core function.

```
=> SELECT page,
 hash_page_type(get_raw_page('t_n_idx',page))
   FROM generate_series(0,3)
 page | hash_page_type
------+----------------
    0 | metapage
    1 | bucket
    2 | bucket
    3 | bitmap
(4 rows)
```

NotebookLM

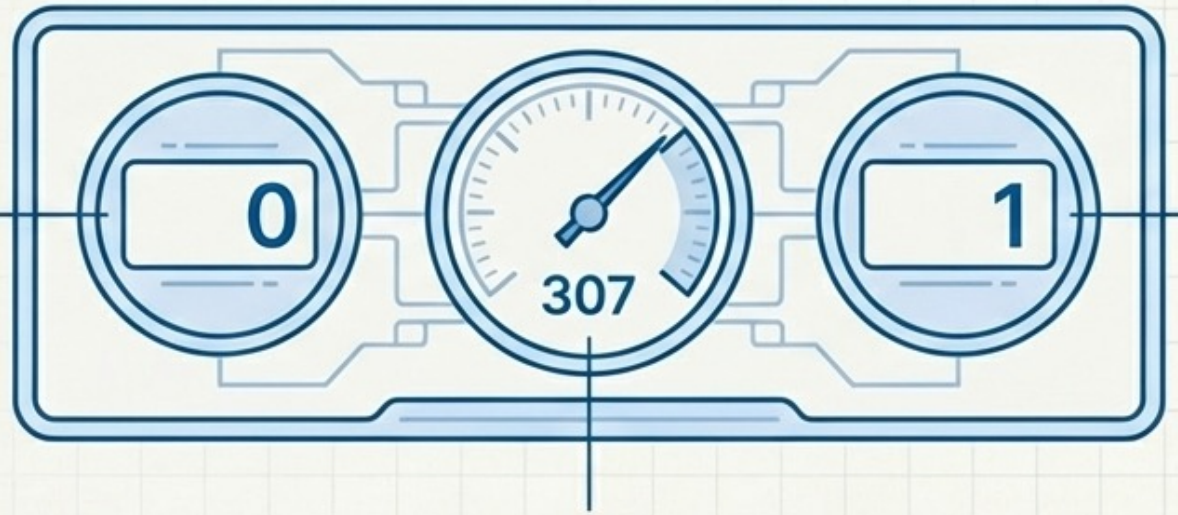
The Control Center: Inside the Metapage.

The metapage contains all control information for the index. Let's examine its key fields.

```
=> SELECT ntuples, ffactor, maxbucket
   FROM hash_metapage_info(get_raw_page('t_n_idx', 0));

  ntuples |  ffactor | maxbucket
----------+----------+-----------
        0 |      307 |         1
(1 row)
```

`ntuples`:
The number of indexed tuples. Currently 0.



`maxbucket`:
The highest bucket number currently in use (zero-indexed). Here, `1` means `w` we have buckets 0 and 1.

`ffactor`

`ffactor`: The target number of tuples per bucket, calculated based on block size and `fillfactor`. In this case, 307.

NotebookLM

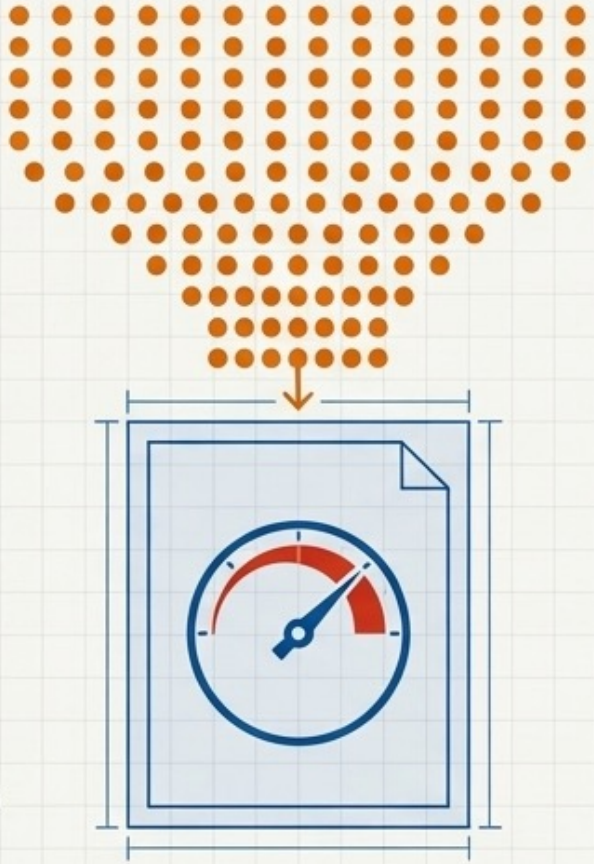
Under Pressure: The Worst-Case Scenario for a Hash Index

A hash index performs best with uniform data distribution. **Its greatest challenge is data skew**, where many identical keys produce the same hash code and target the same bucket.

The Code (The Stress Test)

We insert 500 rows with the **exact same value**.

```
=> INSERT INTO t(n) SELECT 0
  FROM generate_series(1,500);
```



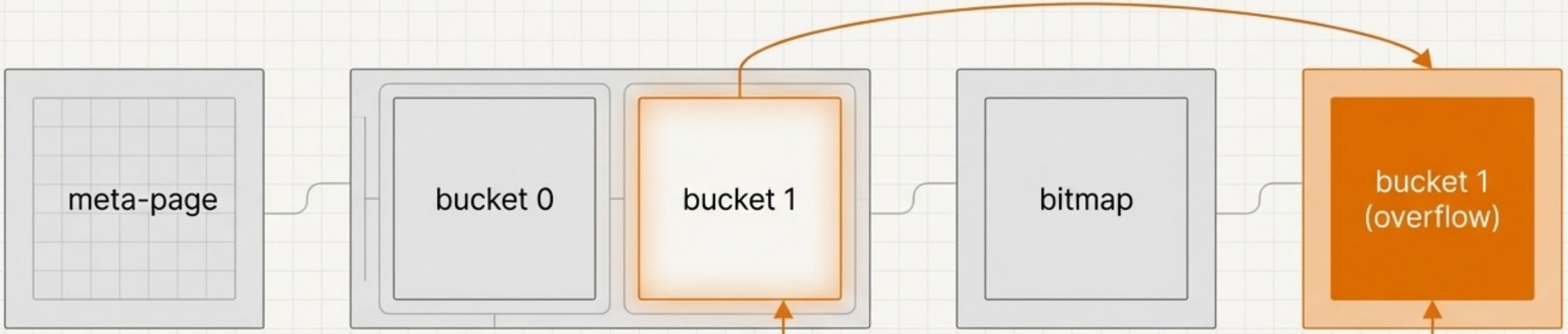
The Consequence

All 500 items target a single bucket. Since this exceeds the `ffactor` (307), the bucket page cannot hold them all. The index is forced to create an **overflow page**.

NotebookLM

The Result of Skew: An Overflow Page is Born

A new page is added to the index to handle the overflow from the full bucket. Performance degrades when a single bucket's items are spread across multiple pages.



The `hash_page_stats` output confirms that bucket 0 is empty, while the 500 items for bucket 1 are split between the primary page (407 items) and the new overflow page (93 items).

```
=> SELECT page, live_items, hasho_bucket
    FROM (VALUES (1), (2), (4)) p(page),
          hash_page_stats(get_raw_page('t_n_idx', page));

  page | live_items | hasho_bucket
-------+------------+--------------
     1 |          0 |            0 -- Bucket 0
     2 |        407 |            1 -- Bucket 1 (Primary)
     4 |         93 |            1 -- Bucket 1 (Overflow)
(3 rows)
```

NotebookLM

Time to Grow: Triggering a Bucket Split.

The hash index expands dynamically when the total number of tuples exceeds the capacity defined by `ffactor`.

Current Tuples (`ntuples`): = 500

Number of Buckets: = 2 (`maxbucket` is 1)

Fill Factor (`ffactor`): = 307

Threshold = **2 * 307 = 614**

The 615th tuple inserted will trigger a split.

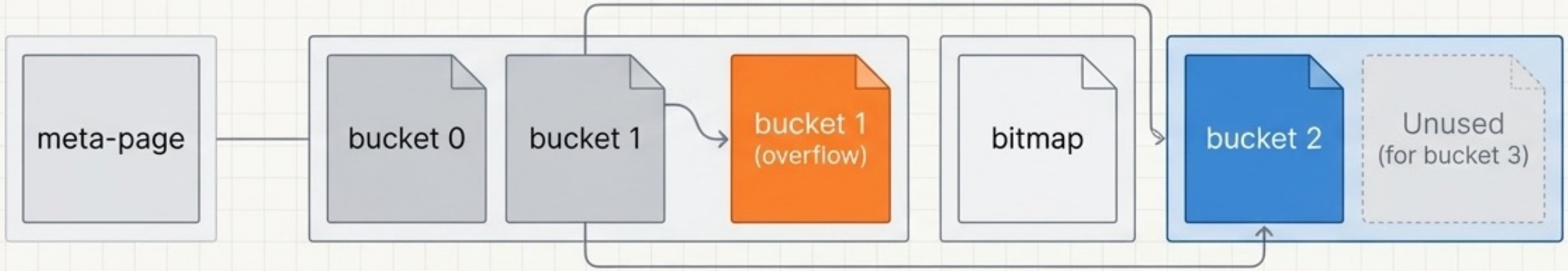
The Code: We insert 115 new, ***distinct*** values to cross the threshold.

```
-- We currently have 500 tuples.
=> INSERT INTO t(n) SELECT n FROM generate_series(1,115);
-- Total tuples is now 615. A split occurs.
```

NotebookLM

Witnessing the Split: Logical Growth vs. Physical Expansion

When a split is triggered, the index grows **logically by one bucket at a time**. However, physically, it allocates a new chunk of pages equal to the current number of buckets.



The Metapage 'Before' and 'After'

Before (ntuples = 500)

```
 ntuples | ffactor | maxbucket
---------+---------+-----------
     500 |     307 |         1
```

After (ntuples = 615)

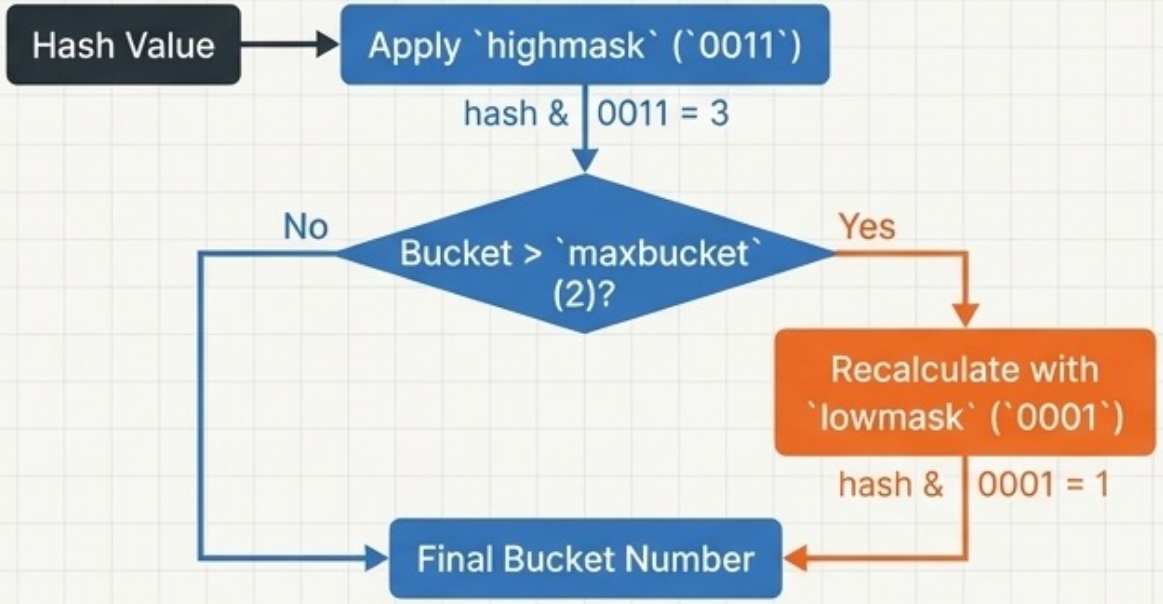
```
 ntuples | ffactor | maxbucket
---------+---------+-----------
     615 |     307 |         2
```

Key Takeaway: The index grows abruptly from the OS perspective but appears to grow incrementally from a logical standpoint.

NotebookLM

The Splitting Algorithm: `highmask` and `lowmask`

How does PostgreSQL map a hash value to a bucket when the number of buckets isn't a power of two (e.g., 3 buckets)? It uses a pair of bitmasks stored in the metapage.



Data from our Example (3 buckets)

```
=> SELECT maxbucket, highmask::bit(4), lowmask::bit(4)
    FROM hash_metapage_info(get_raw_page('t_n_idx', 0));

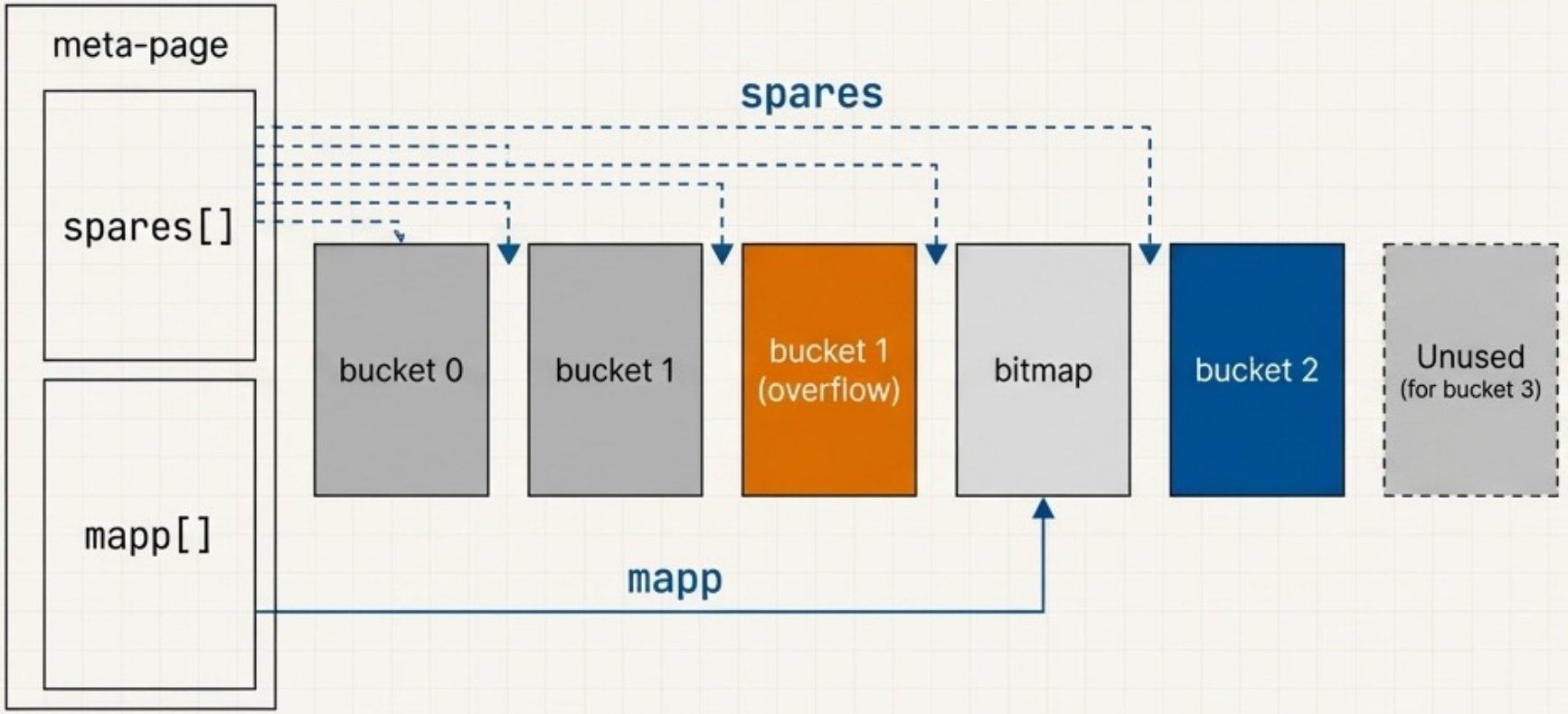
 maxbucket | highmask | lowmask
-----------+----------+---------
         2 |     0011 |    0001
```

In this case, a hash resulting in bucket `3` (binary `11) would be re-mapped to bucket `1` (binary `01) using the `lowmask`.

NotebookLM

Navigating the File: `spares` and `mapp` arrays

The metapage stores arrays to help locate pages quickly, accounting for the overflow and bitmap pages that are interspersed between bucket page blocks.



`spares` array

Records the cumulative number of non-bucket pages (like overflow, bitmap) inserted *before* each doubling of the index size. This allows for simple arithmetic to find a bucket's block number.

`mapp` array

Contains direct pointers to the block numbers of the bitmap pages themselves.

NotebookLM

Housekeeping: Reclaiming Space and a Key Limitation.



Reclaiming Space within Pages

Space from deleted tuples is reclaimed during page pruning (triggered on insert attempts to a full page) or by a standard `VACUUM`.



Reusing Pages

Cleared overflow pages are tracked in the bitmap and can be reused by any bucket that needs one.

Primary bucket pages are permanently assigned to their bucket, even if empty.

The Critical Limitation: Hash Indexes Do Not Shrink.



Once allocated, pages are never returned to the operating system.

To reduce the physical size of a bloated hash index, you must rebuild it entirely using `REINDEX` or `VACUUM FULL`.

NotebookLM

Capabilities Scorecard: Access Method Properties.

What can the hash index method do? Here are its core capabilities.

Ordering & Uniqueness		Structure	
✗	`can_order`: ✗ **No**. The hash function randomizes data order.	✗	`can_multi_col`: ✗ **No**. Hash indexes are single-column only.
✗	`can_unique`: ✗ **No**. A `UNIQUE` constraint is not supported directly.	✗	`can_include`: ✗ **No**. `INCLUDE` columns are not supported.
✓	`can_exclude`: ✔ **Yes**. You can use an `EXCLUDE USING hash (... WITH =)` constraint to enforce uniqueness.		

NotebookLM

Capabilities Scorecard: Scans and Column Properties.

A look at supported scan types and column-level behaviors.

Index Scan Properties

✓	`index_scan`: ✔ **Yes**. Both regular Index Scans and Bitmap Index Scans are fully supported.
✗	`clusterable`: ✕ **No**. It makes no sense to physically order a table by hash values.

Column Level Properties

✗	`returnable` (Index-Only Scans): ✕ **No**. The index does not store the original key value, so a heap fetch is always required to verify the key and return the value.
✗	`search_nulls`: ✕ **No**. The equality operator (`=`) is not defined for `NULL` values, so they cannot be indexed.

NotebookLM

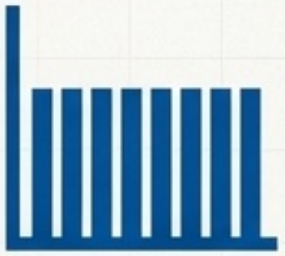
The Hash Index: A Specialist's Tool

Use For



Blazing-fast, single-column equality lookups (= in JetBrains Mono)

Ideal Conditions / Beware Of



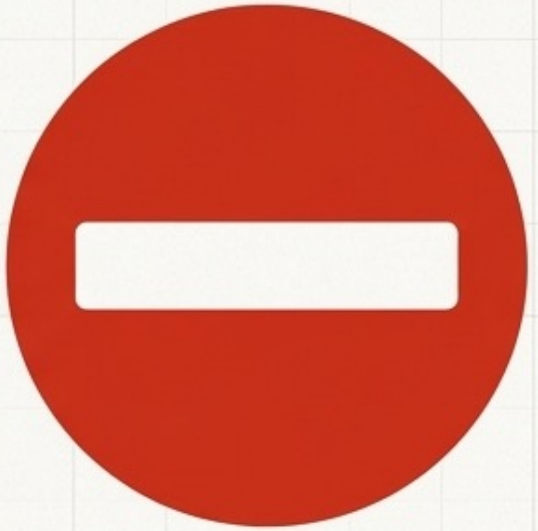
Ideal Conditions (in Inter Bold): Data with high cardinality and a uniform distribution.



Beware Of (in Inter Bold): Highly skewed data or many duplicate keys, which lead to performance-killing overflow pages.

The fact that they grow in large chunks and **do not shrink** (in Inter Bold). Plan to `REINDEX` (in JetBrains Mono) periodically if the table has high churn.

Avoid For



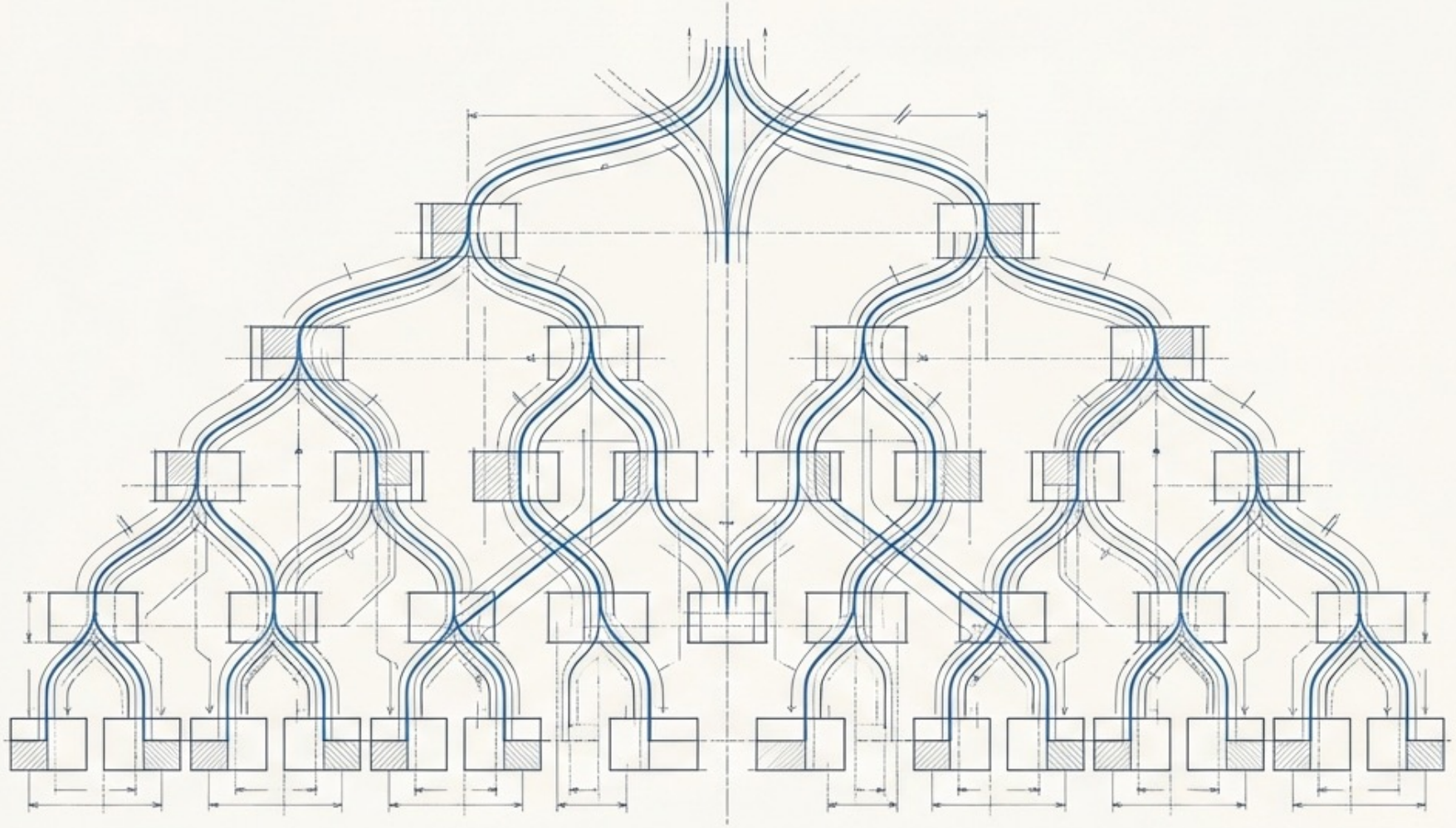
Range queries (>, < in JetBrains Mono), ordering (`ORDER BY` in JetBrains) multi-column logic, or index-only scans.

NotebookLM

25. B-tree

PostgreSQL B-Trees: From Blueprint to Masterpiece

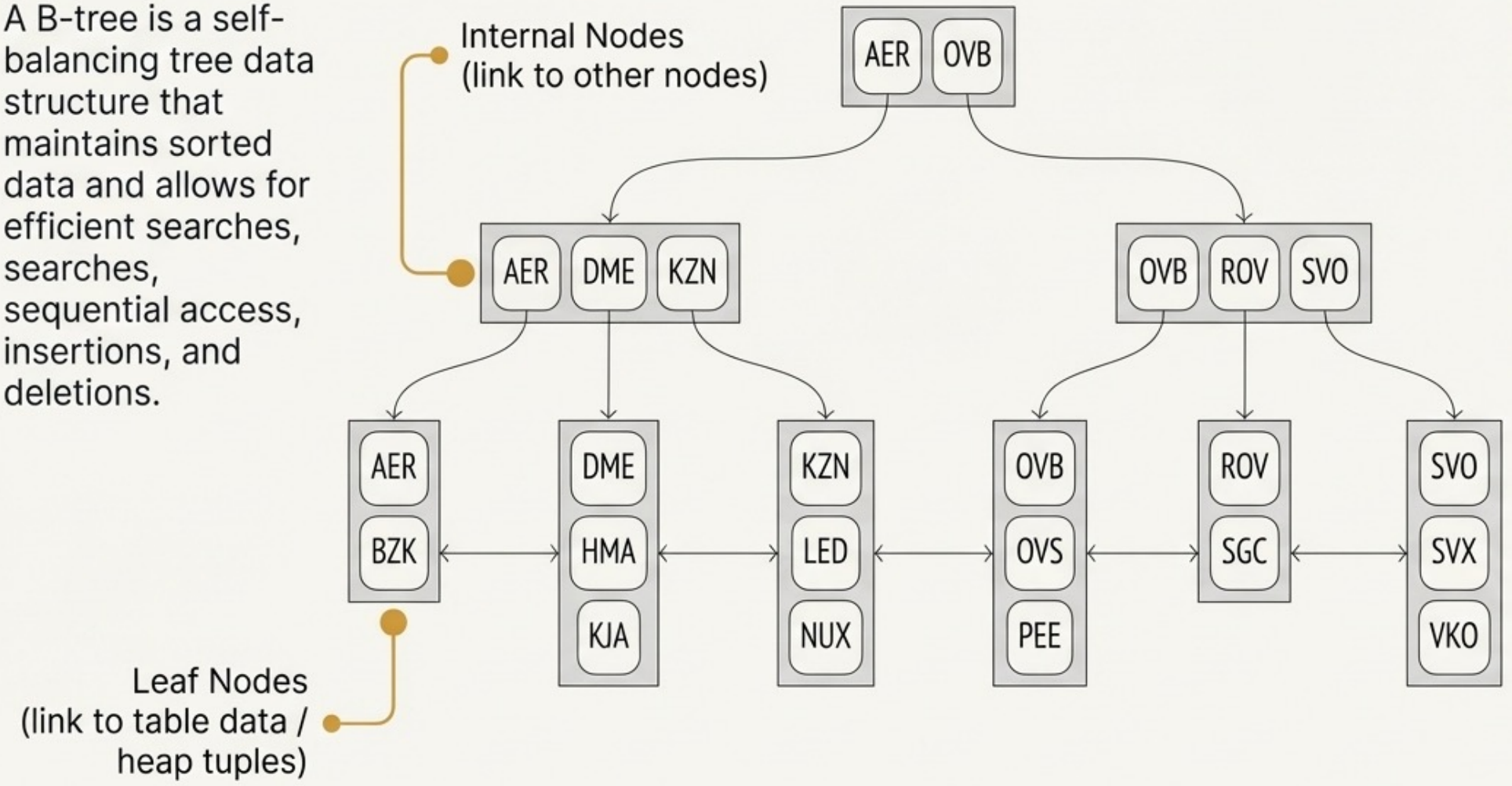
An in-depth look at the structure, implementation, and optimization of PostgreSQL's most fundamental index.



NotebookLM

The Anatomy of a B-Tree

A B-tree is a self-balancing tree data structure that maintains sorted data and allows for efficient searches, searches, sequential access, insertions, and deletions.



Balanced: All leaf nodes are at the same depth, ensuring consistent search performance for any value.

Bushy (High Fanout): Each node contains many keys (often hundreds), resulting in a very shallow tree even for massive tables.

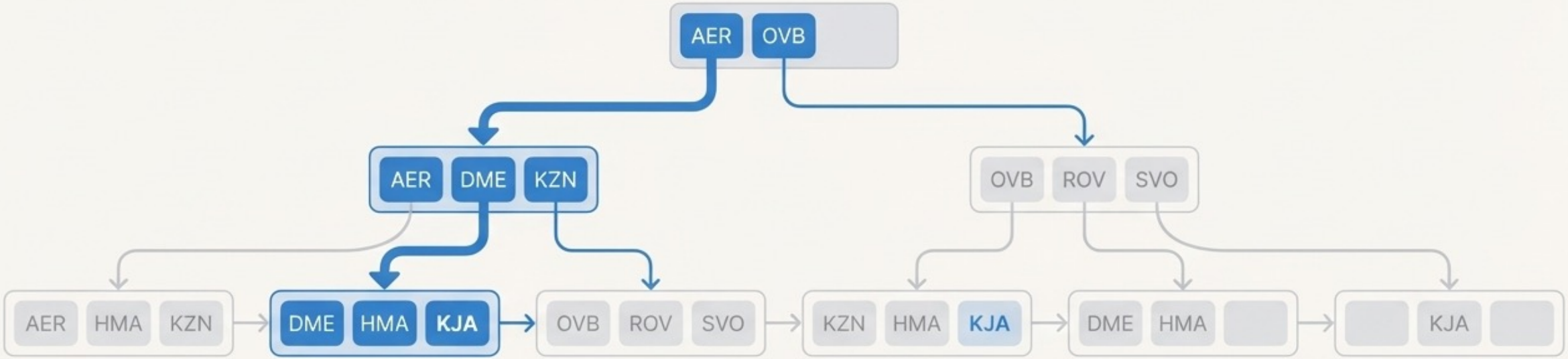
Sorted: Data is sorted within each node and across all nodes at the same level. Leaf nodes are connected via a doubly-linked list for efficient ordered scans.

NotebookLM

Finding Your Way: The Search for Equality

Scenario: We need to find the airport code 'KJA'.

The Process: The search begins at the root and recursively descends the tree. At each internal node, the algorithm selects the child pointer between keys K_i and K_{i+1} that satisfies K_i <= 'KJA' < K_{i+1}.



Root Node

Contains `AER` and `OVB`. Since `AER <= 'KJA' < OVB` is true, we follow the pointer from `AER`.

Internal Node

Contains `AER` and `DME`, `KZN`. Since `DME <= 'KJA' < KZN` is true, we follow the pointer from `DME`.

Leaf Node

We arrive at the correct leaf node containing `DME`, `HMA`, `KJA` and find our value.

Note: B-trees optimize space by not storing the redundant leftmost key in internal nodes (e.g., `AER` in the second level).

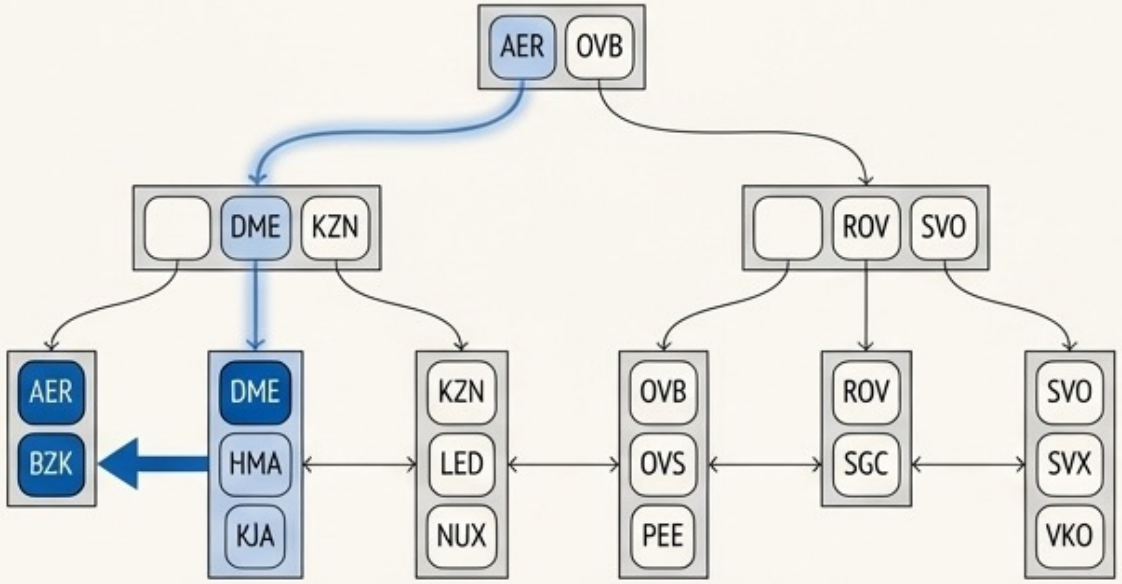
NotebookLM

Scanning Beyond a Single Point: Inequalities and Ranges

The doubly-linked list connecting all leaf nodes allows for efficient traversal without repeatedly descending from the root.

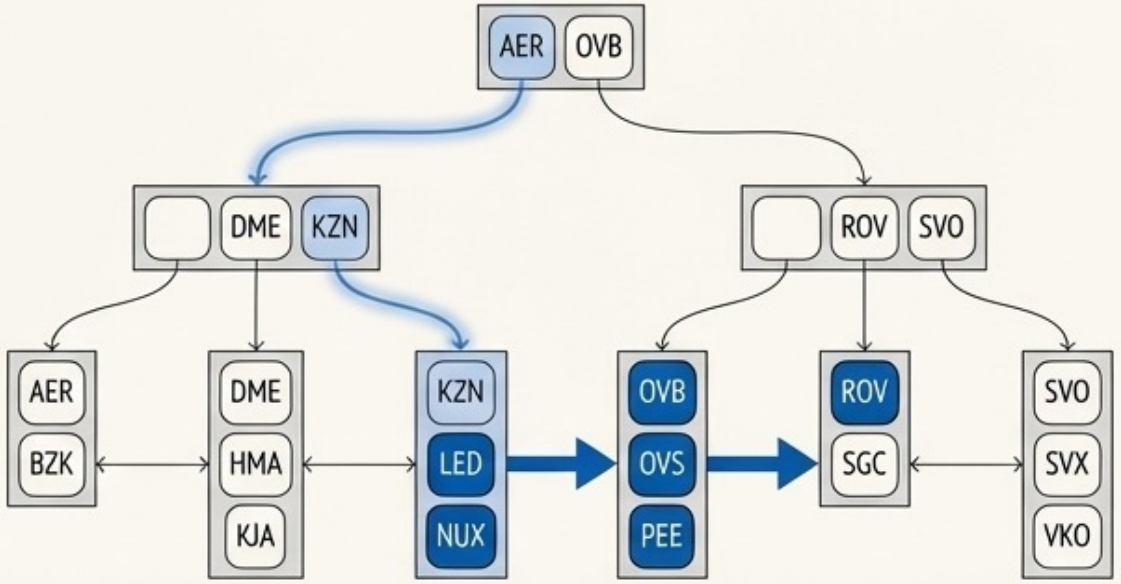
Inequality Scan (`<= 'DME'`)

1. First, the tree is traversed to locate the value `DME'.
2. Then, the search scans backward (left) along the linked list, collecting all preceding values.



Range Scan (`BETWEEN 'LED' AND 'ROV'`)

1. The tree is traversed to find the starting value, `LED'.
2. The search then scans forward (right) along the linked list, collecting values until it passes `ROV'.



NotebookLM

Making Room: How B-Trees Grow Through Insertions and Splits

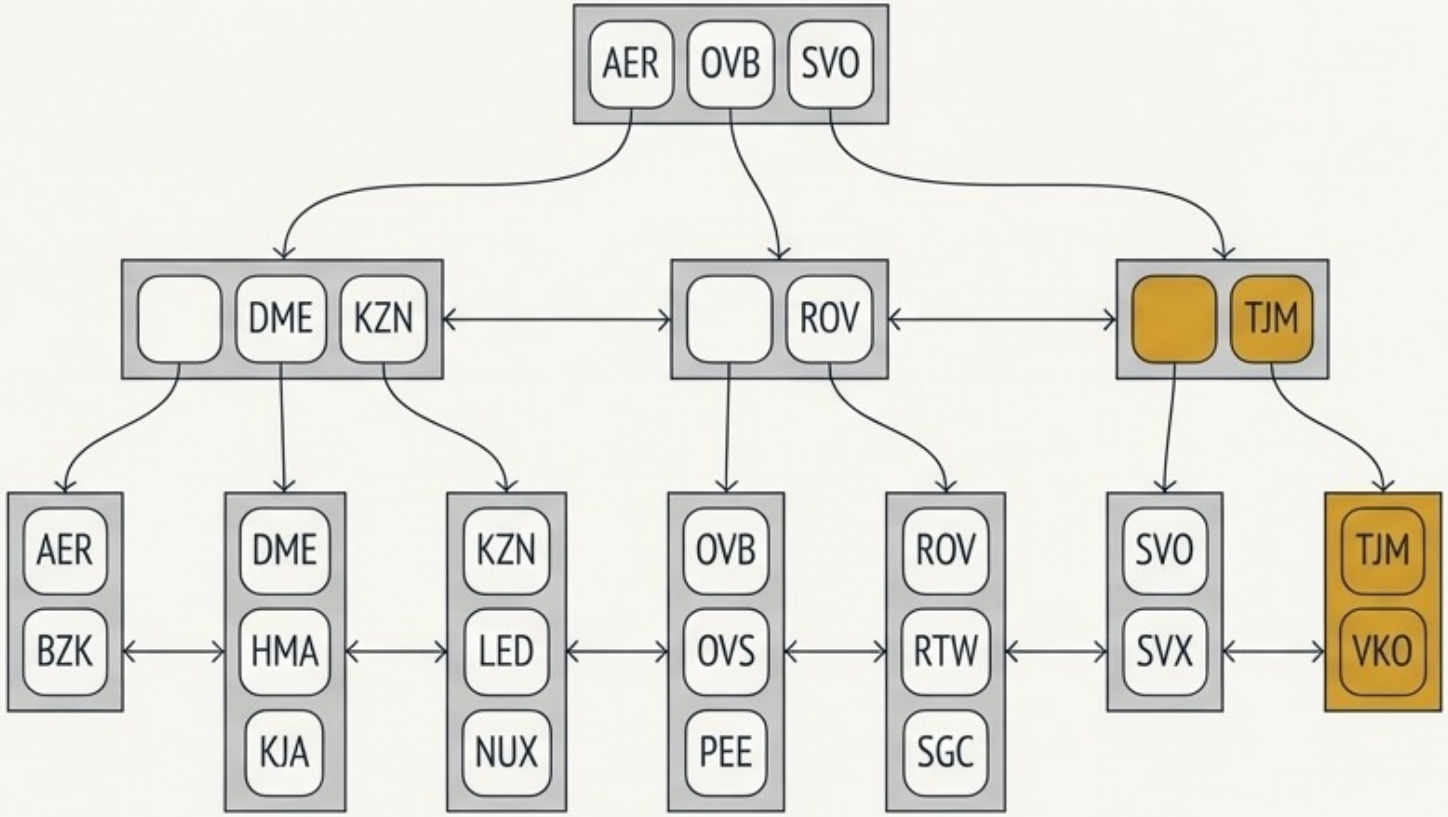
Process:

1. **Insertion**: A new key's location is uniquely determined by the sort order. E.g., 'RTW' is inserted between 'ROV' and 'SGC'.
2. **Node Splitting**: If the target leaf node is full, it splits in two. Half the elements move to a new node.
3. **Parent Update**: A pointer to the new node is added to the parent node.
4. **Cascading Splits**: If the parent is also full, it splits as well. This can propagate all the way to the root. If the root splits, a new root is created, and the tree's depth increases by one.

Example: Inserting 'TJM' when the final leaf node is full, causing a leaf split and an internal node split.

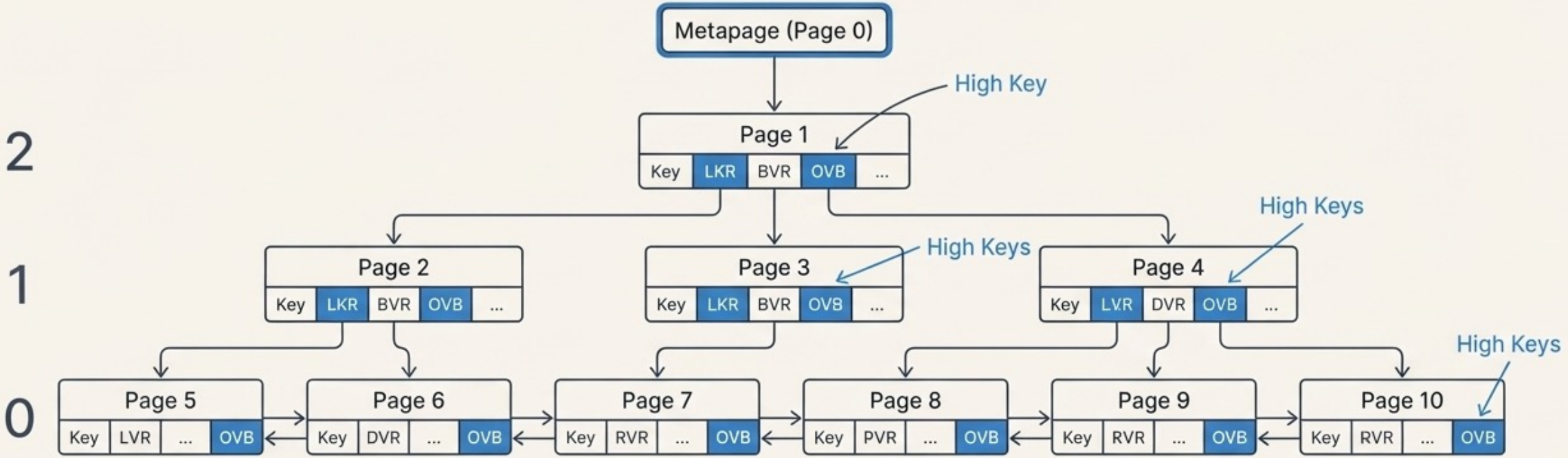
PostgreSQL Specifics:

PostgreSQL's B-tree implementation does not merge pages back together after deletions. To avoid splitting, it will first attempt to free space on a full page by removing obsolete data.



NotebookLM

From Nodes to Pages: The Physical On-Disk Structure



Core Concepts

- **One Node, One Page**": Each node in the B-tree corresponds to a single disk page (typically 8kB).
- **The Metapage**": Page 0 of the index is the metapage. It's a fixed entry point that stores metadata, including the block number of the current root page and the depth of the tree.
- **Tree Levels**": Levels are numbered starting from 0 for the leaf nodes.
- **High Keys**": To aid in concurrent operations, every page (except the rightmost at each level) stores a "high key"—a value guaranteed to be greater than or equal to all other keys on that page.

NotebookLM

A Guided Tour with `pageinspect`, Part 1: Descending the Tree

Goal: Find the booking with `book_ref = 'E2D725'`.

Step 1: Inspect the Metapage

Find the root page and tree depth.

```
=> SELECT root, level FROM bt_metap('bookings_pkey');
```

```
 root | level
------+-------
  290 |     2
```

Step 2: Inspect the Root Page (Page 290)

Use a helper function (`data_to_text`) to decode the keys. We look for the entry where `key <= 'E2D725' < next_key`.

```
=> SELECT itemoffset, ctid, data_to_text(data)
    FROM bt_page_items('bookings_pkey', 290);
```

```
 itemoffset |   ctid   | data_to_text
------------+----------+--------------
 ...        |          |
         19 | (5135,1) | E2CB14
         20 | (5420,1) | EF6FEA
 ...        |          |
```

Decision

The condition `E2CB14 <= 'E2D725' < EF6FEA` is met at `itemoffset 19`. We follow its `ctid` pointer to page **5135**.

NotebookLM

A Guided Tour with `pageinspect`, Part 2: Reaching the Leaf

Step 3: Inspect the Intermediate Page (Page 5135)
This page contains a high key at the start. We repeat the search process.

```
=> SELECT itemoffset, ctid, data_to_text(data)
     FROM bt_page_items('bookings_pkey', 5135);

 itemoffset |   ctid   | data_to_text
------------+----------+-------------
          1 | (5417,1) | EF6FEA      -- High Key
 ...
          3 | (5133,1) | E2D71D
          4 | (5134,1) | E2E2F4
 ...
```

Decision: We follow the pointer from `itemoffset` 3 to page **5133**.

Step 4: Inspect the Leaf Page (Page 5133)
Here, the `ctid`s point to actual table rows. We find our exact match.

```
-- On page 5133...
   3 | (11919,77) | E2D725
```

Step 5: Fetch the Row: `SELECT * FROM bookings WHERE ctid = '(11919,77)'`. This returns the full booking record.



The Abstraction: All these steps are what happens behind this simple plan:

```
 QUERY PLAN
-----------------------------------------
Index Scan using bookings_pkey on bookings
  Index Cond: (book_ref = 'E2D725'::bpchar)
```

NotebookLM

The Art of Compression: Deduplication

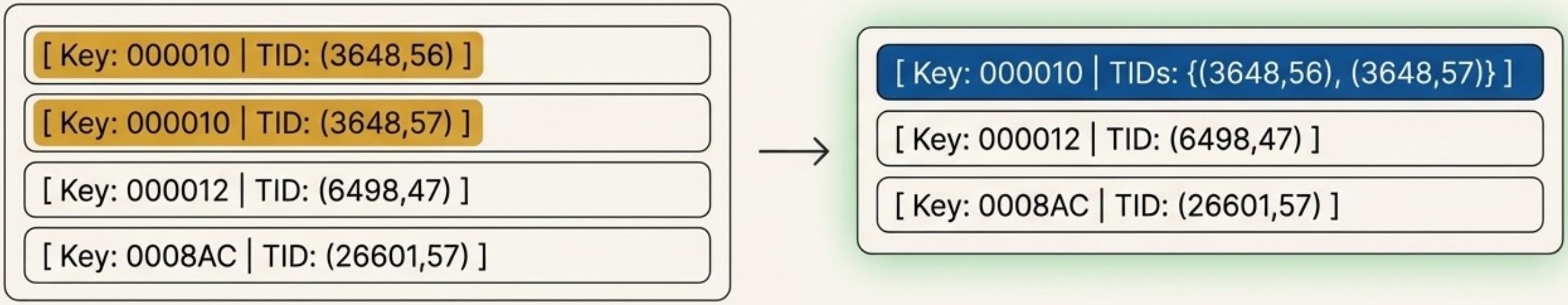
The Problem

Non-unique indexes can contain many duplicate keys pointing to different rows. Even unique indexes can have duplicates due to MVCC (multiple row versions). These duplicates consume significant space.

The Solution

Instead of storing the key multiple times, deduplication collapses these entries into a single index item containing the key and a list of tuple IDs (TIDs).

Before & After



How it's Triggered

Deduplication is not performed proactively. It runs only when an insertion is attempted on a page that is full, as a last resort to create space and avoid a page split.

Benefits

- Significantly reduces index size for columns with low cardinality.
- Helps mitigate index bloat from MVCC.
- Avoids unnecessary page splits, keeping the tree shallower.

NotebookLM

Deduplication in Practice (and Its Limits)

Checking for Support

The `allequalimage` field in the metapage indicates if deduplication is supported for an index.

```
=> SELECT allequalimage FROM bt_metap('tickets_book_ref_idx');
 allequalimage
---------------
             t
```

Inspecting a Deduplicated Page

A page with deduplication will have a mix of items with a single tuple ID (`htid`) and items with a list of IDs (`tids`).

```
=> SELECT itemoffset, htid, tids::text
    FROM bt_page_items('tickets_book_ref_idx', 1) WHERE ...;

 itemoffset |    htid    |            tids
------------+------------+----------------------------
          3 | (47429,51) |
          4 | (3648,56)  | {"(3648,56)","(3648,57)"}
```

Key Limitations

Deduplication is not supported for:

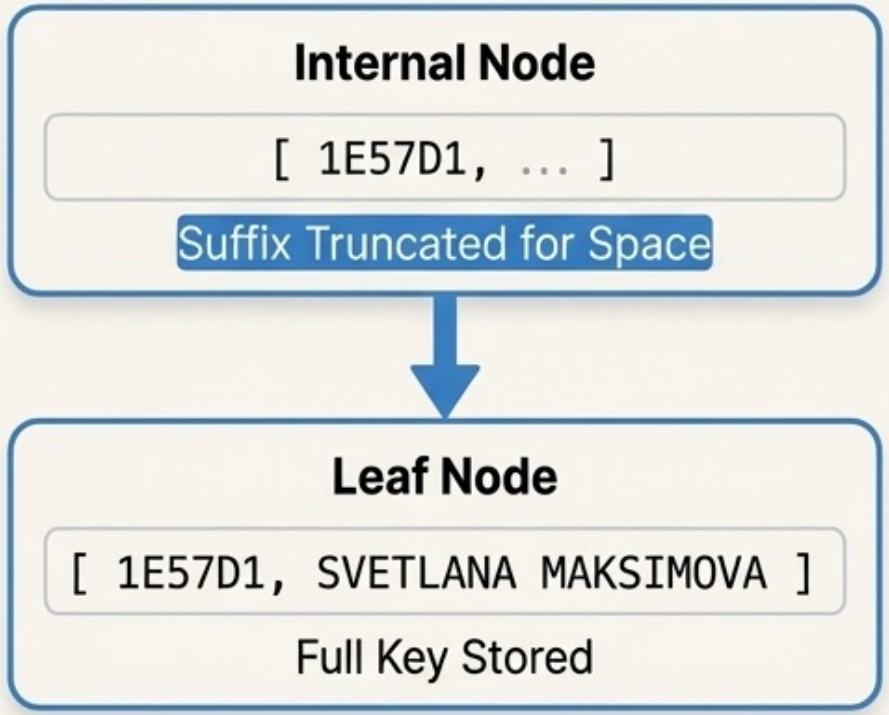
- Data types where equality cannot be determined by a simple binary comparison (e.g., `float`, `numeric`, `jsonb`).
- Types with non-deterministic collations.
- Composite types, ranges, and arrays.
- Indexes with an `INCLUDE` clause.

NotebookLM

Keeping it Lean: Suffix Truncation in Internal Nodes

The Goal
Internal node entries only exist to direct search traffic. They don't need to store the full key value, only enough information to distinguish between the child sub-trees.

The Technique
For multi-column indexes, PostgreSQL can truncate trailing columns from keys stored in internal pages. The first column (or first few) is often sufficient to make the routing decision.



Example*: An index on `(book_ref, passenger_name)`. In the root page, some entries may only store the `book_ref`, omitting the `passenger_name` to save space.

```
-- Root page of tickets_bref_name_idx
=> SELECT itemoffset, data_to_text(data) FROM ...;
 itemoffset | data_to_text
------------+------------------------------
          8 | 1A98A0
          9 | 1E57D1, SVETLANA MAKSIMOVA
         10 | 220797
```

Important Note: Leaf pages must always store the full key values to support index-only scans.

NotebookLM

Defining the Rules of Order: Operator Classes

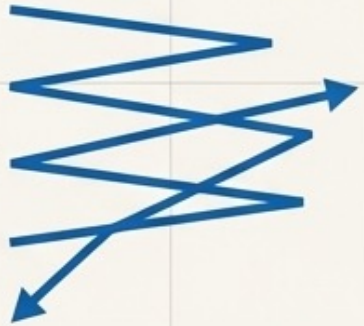
The Challenge

How does a B-tree know how to sort custom data types? Default sorting (lexicographical) is often not what's needed.

Default (Incorrect) Sort

A custom `capacity` type stores `(amount, unit)`. Lexicographical sorting is incorrect:

(1,B)
(3,GB)
(4,MB)
(9,kB)
...



Desired (Natural) Sort

We need a natural sort that understands the units:

(1,B)
(9,kB)
(4,MB)
(3,GB)
...



The Solution

An **Operator Class** is a collection of operators and support functions that defines comparison semantics for a specific data type and index method. For B-trees, this means defining:

- 5 Comparison Operators: `<`, `<=`, `=`, `>=`, `>`
- 1 Core Support Function: A `compare(a, b)` function that returns -1, 0, or 1.

NotebookLM

Case Study: Building an Operator Class for `capacity`

Step 1: Create a Comparison Function

This function converts values to a common unit (bytes) and compares them.

```
CREATE FUNCTION capacity_cmp(a capacity, b capacity) RETURNS integer AS $$
  SELECT sign(capacity_to_bytes(a) - capacity_to_bytes(b));
$$ ...;
```

Step 2: Define Operators

Create the five required operators that use the comparison function.

```
CREATE FUNCTION capacity_lt(a capacity, b capacity) RETURNS boolean ...;
CREATE OPERATOR #<# (FUNCTION = capacity_lt, ...);
-- (Repeat for <=, =, >=, >)
```

Step 3: Create the Operator Class

Bundle the operators and function into a named class for the `btree` access method.

```
CREATE OPERATOR CLASS capacity_ops DEFAULT FOR TYPE capacity USING btree AS
  OPERATOR 1 #<#, ...
  FUNCTION 1 capacity_cmp(capacity, capacity);
```

The Result

Queries now use the correct, natural sort order, and an index on `cap` can efficiently serve queries like.

```
WHERE cap #<# '(100,B)'::capacity
```

NotebookLM

The Cardinal Rule: Column Order is Everything

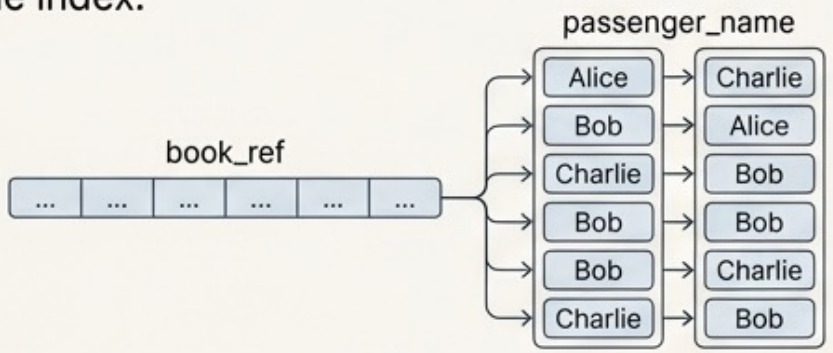
The Principle: A multi-column index on `(col_A, col_B)` is sorted first by `col_A`, and then by `col_B` within each `col_A` value. This structure dictates which queries can use the index efficiently.

Case 1: `INDEX ON tickets(book_ref, passenger_name)`

✅ **Efficient**: `WHERE book_ref = '...'`

✅ **Efficient**: `WHERE book_ref = '...' AND passenger_name = '...'`

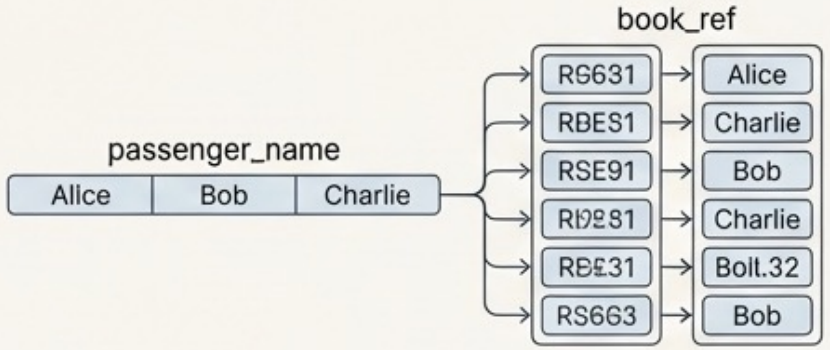
❌ **Inefficient**: `WHERE passenger_name = '...'` → Leads to a full table scan, as passenger names are scattered throughout the index.



Case 2: `INDEX ON tickets(passenger_name, book_ref)`

✅ **Efficient**: `WHERE passenger_name = '...'` → Uses a Bitmap Index Scan.

❌ **Inefficient**: `WHERE book_ref = '...'` → Would be inefficient.



Key Takeaway

Place columns in the index in the order they are most frequently used in `WHERE` clauses, starting with the most selective columns.

NotebookLM

Designing the Perfect Index for Sorting

Beyond Filtering: B-tree indexes can also satisfy `ORDER BY` clauses, eliminating the need for a separate, costly sort step in the query plan.

The Challenge

A query needs a specific sort order: `ORDER BY passenger_name ASC, book_ref DESC`. A standard index on `(passenger_name, book_ref)` (which is `ASC, ASC` by default) cannot satisfy this.

```
-- Using a standard index results in a post-scan sort
QUERY PLAN
------------------------------------------------------
Incremental Sort
  Sort Key: passenger_name, book_ref DESC
  -> Index Scan using tickets_name_bref_idx ...
```

The Masterpiece Index

By creating an index that precisely matches the required ordering, including direction and null placement, we can get a direct index scan.

```
-- Create the perfect index
CREATE INDEX tickets_name_bref_idx2
ON tickets(passenger_name ASC NULLS FIRST, book_ref DESC);
```

```
-- The plan now uses the index directly, no sort needed
QUERY PLAN
------------------------------------------------------
Index Scan using tickets_name_bref_idx2 on tickets
```

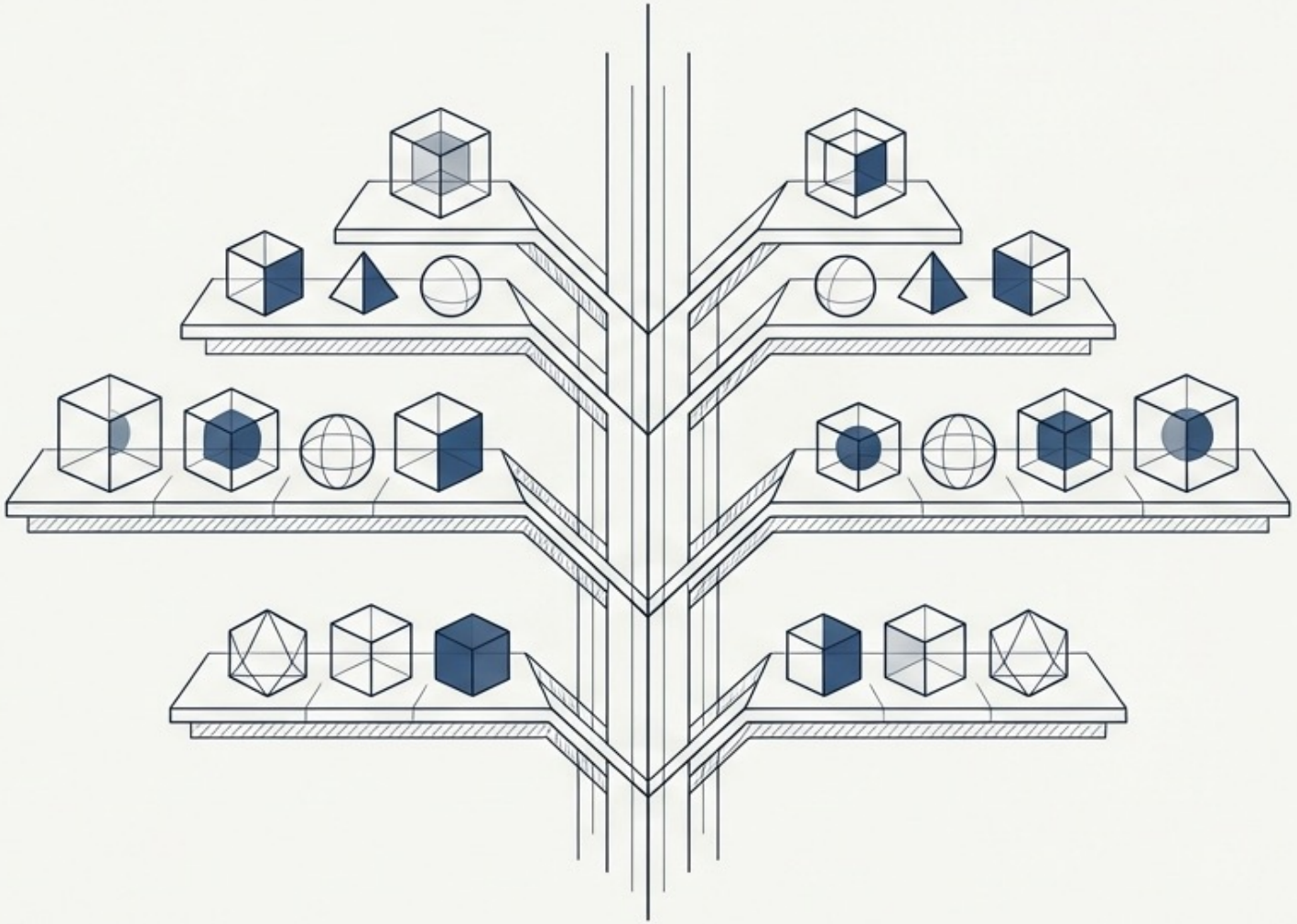
A well-designed B-tree is more than a search tool; it is a critical component of query performance that can pre-calculate the exact order of results.

NotebookLM

26. GiST

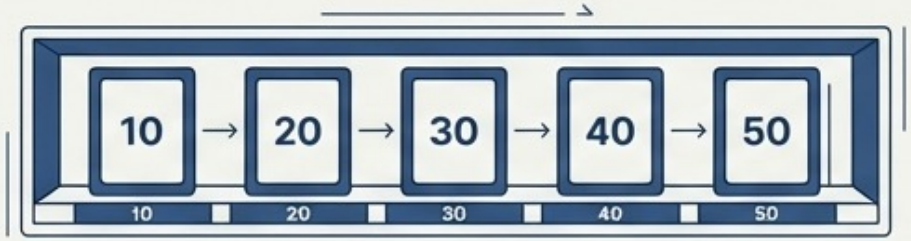
GiST: The Generalized Search Tree Framework

Unlocking High-Performance Indexing for Complex Data in PostgreSQL



NotebookLM

When Standard B-Trees Aren’t Enough



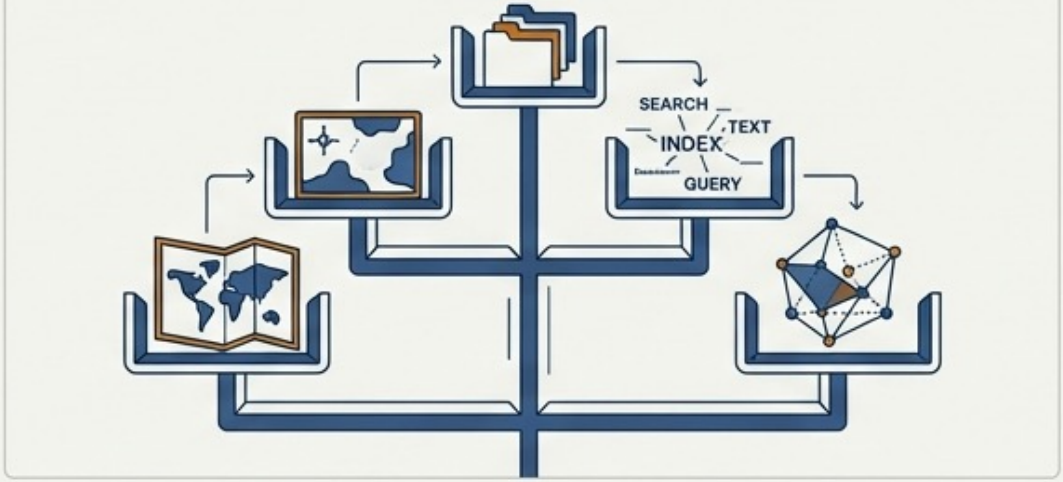
B-tree: The Sorted Filing Cabinet

Highly efficient for standard, orderable data types (integers, text, dates). Relies on comparison operators (<, =, >).

Analogy

Like a perfectly sorted filing cabinet for paper documents. Excellent for finding a specific document or a range of documents.

Limitation: What about data that doesn’t have a simple linear order?



GiST: The Modular Shelving System

A generalized, balanced search tree framework. Designed for complex, non-scalar data types where relative position is key.

Examples

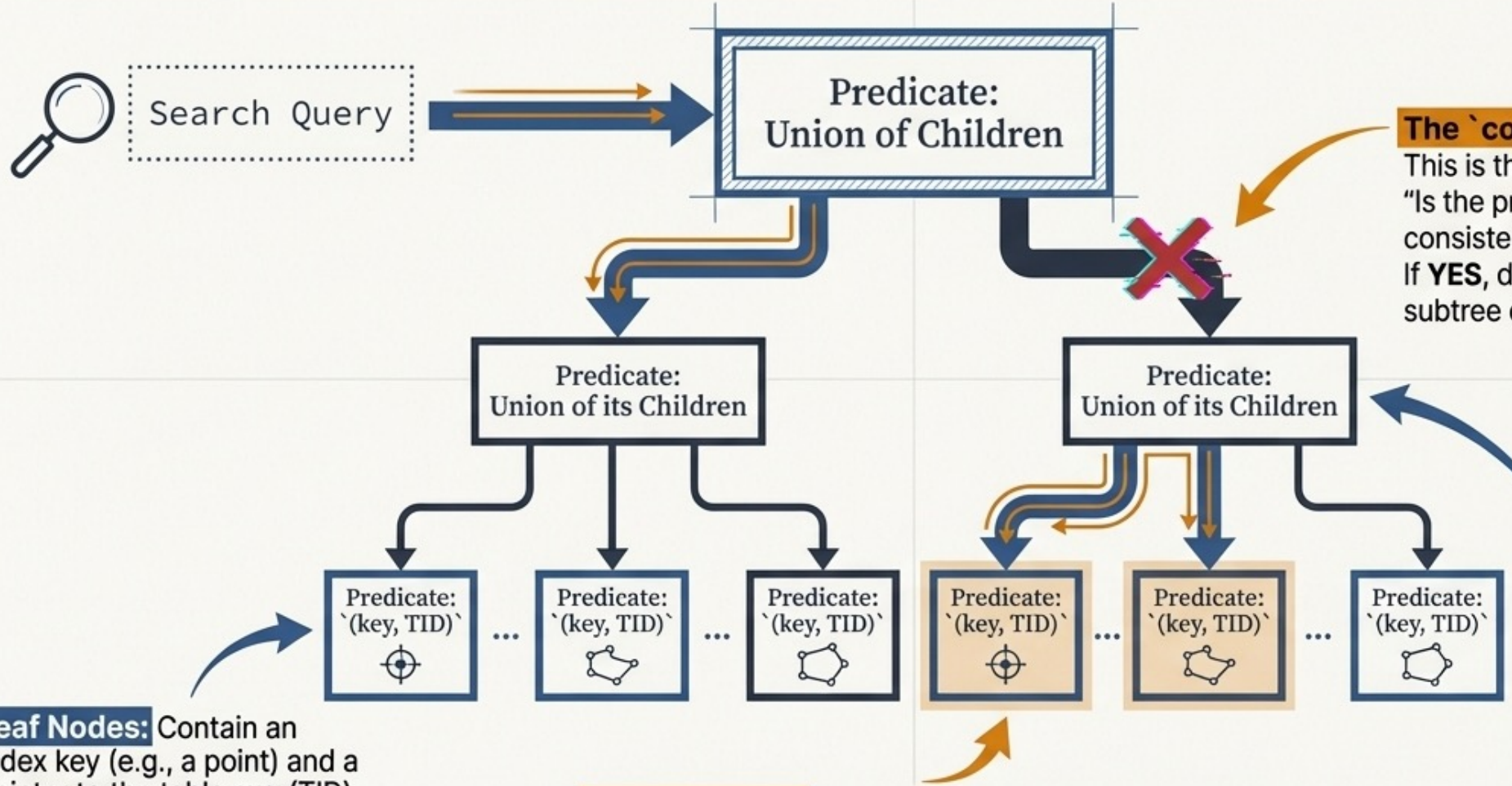
Spatial data (points, polygons), sets of words (full-text search), network addresses, geometric ranges.

Core Idea

GiST provides the low-level infrastructure (page layout, locking, WAL), allowing developers to define custom indexing logic via an ‘`operator class`.’

NotebookLM

The Core Principle: A Tree of "Consistent" Predicates



The `consistency` function: This is the heart of GiST. It asks: "Is the predicate of this index entry consistent with my search query?" If **YES**, descend. If **NO**, the entire subtree can be pruned.

Internal Nodes: Do NOT contain simple ranges. Instead, they hold a predicate that represents the *union* of all child node predicates. For example, a bounding box that contains all the points in its subtree.

Leaf Nodes: Contain an index key (e.g., a point) and a pointer to the table row (TID). The key satisfies a predicate (e.g., 'point is at coordinate X,Y').

Key Difference: Unlike a B-tree, a GiST search might need to traverse multiple branches from a single node if their predicates overlap with the search query.

NotebookLM

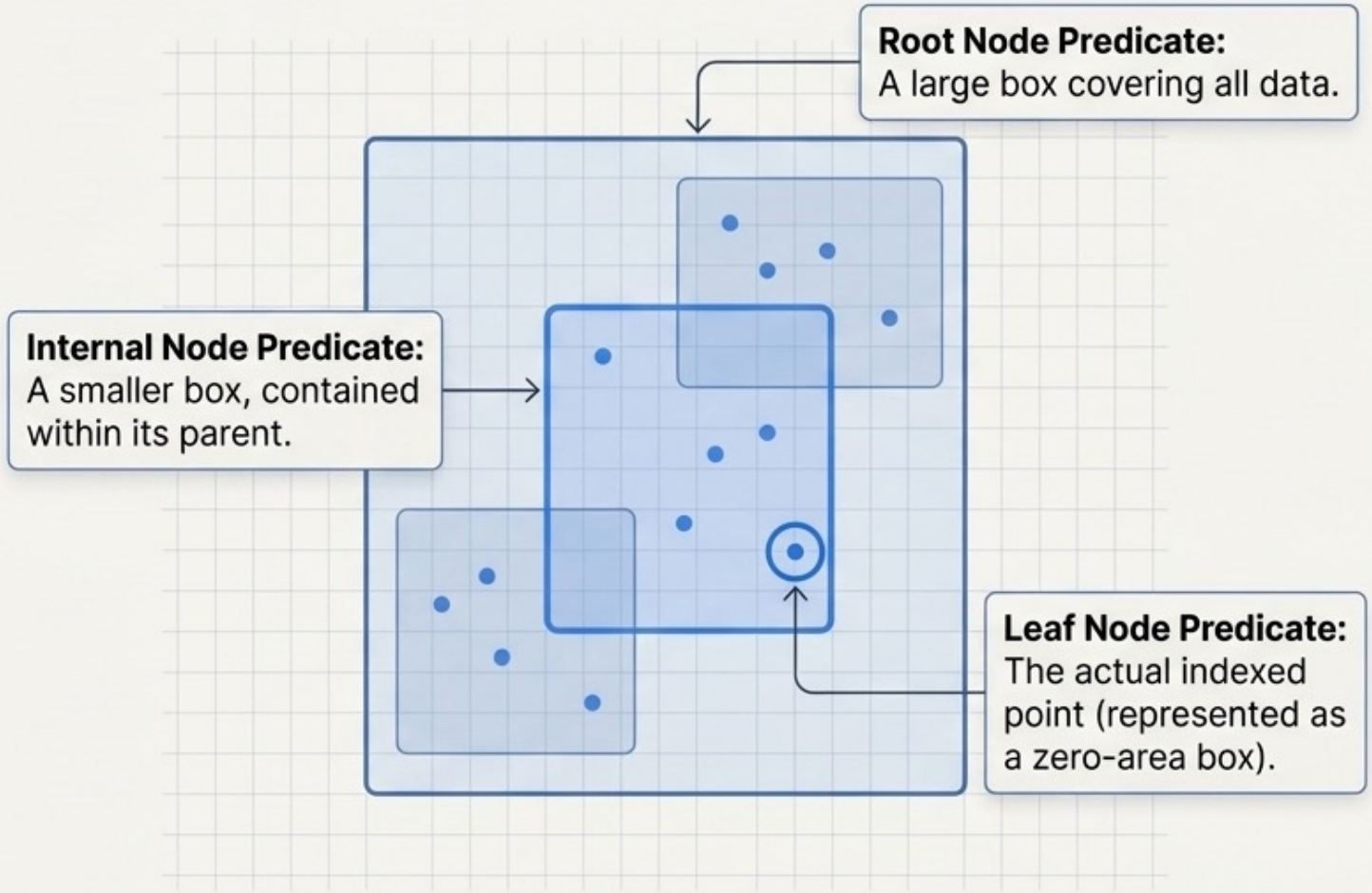
Case Study 1: R-trees for Spatial Data

The Problem

How do you efficiently index 2D data like geographical coordinates?
A B-tree can't sort points in a meaningful way for proximity or containment queries.

The R-tree Solution

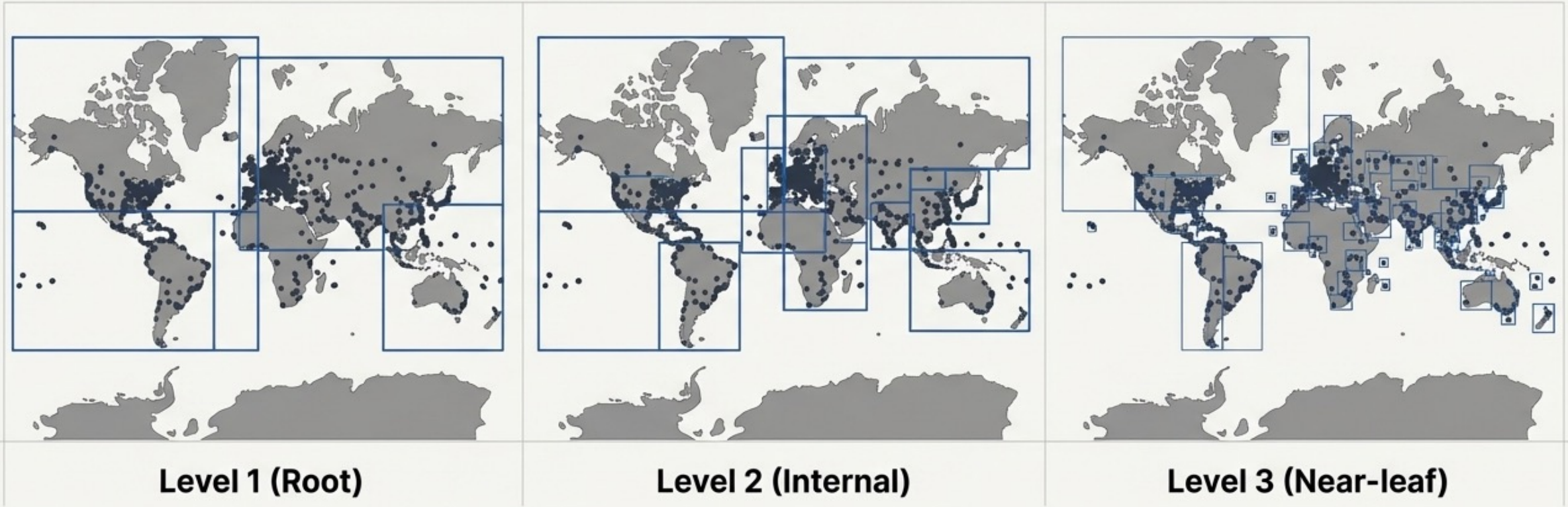
- It's a GiST implementation for spatial data.
- **Predicate**: "The point is contained within this bounding box."
- **Hierarchy**: The tree is a hierarchy of nested bounding boxes.



NotebookLM

Visualizing the R-tree Hierarchy for Airport Coordinates

An R-tree built on a table of ~5,000 airport coordinates. The `fillfactor` was lowered to create a deeper tree for demonstration.



Level 1 (Root) | Level 2 (Internal) | Level 3 (Near-leaf)

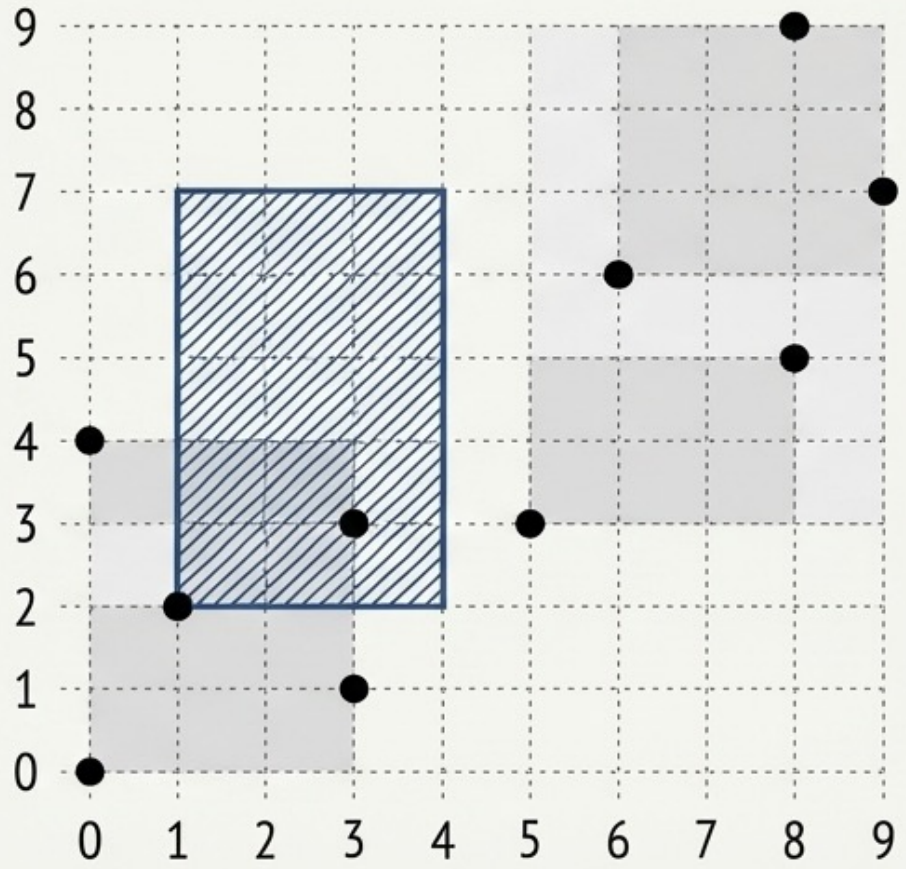
Each level refines the spatial partitioning, allowing the index to quickly narrow down the search area.

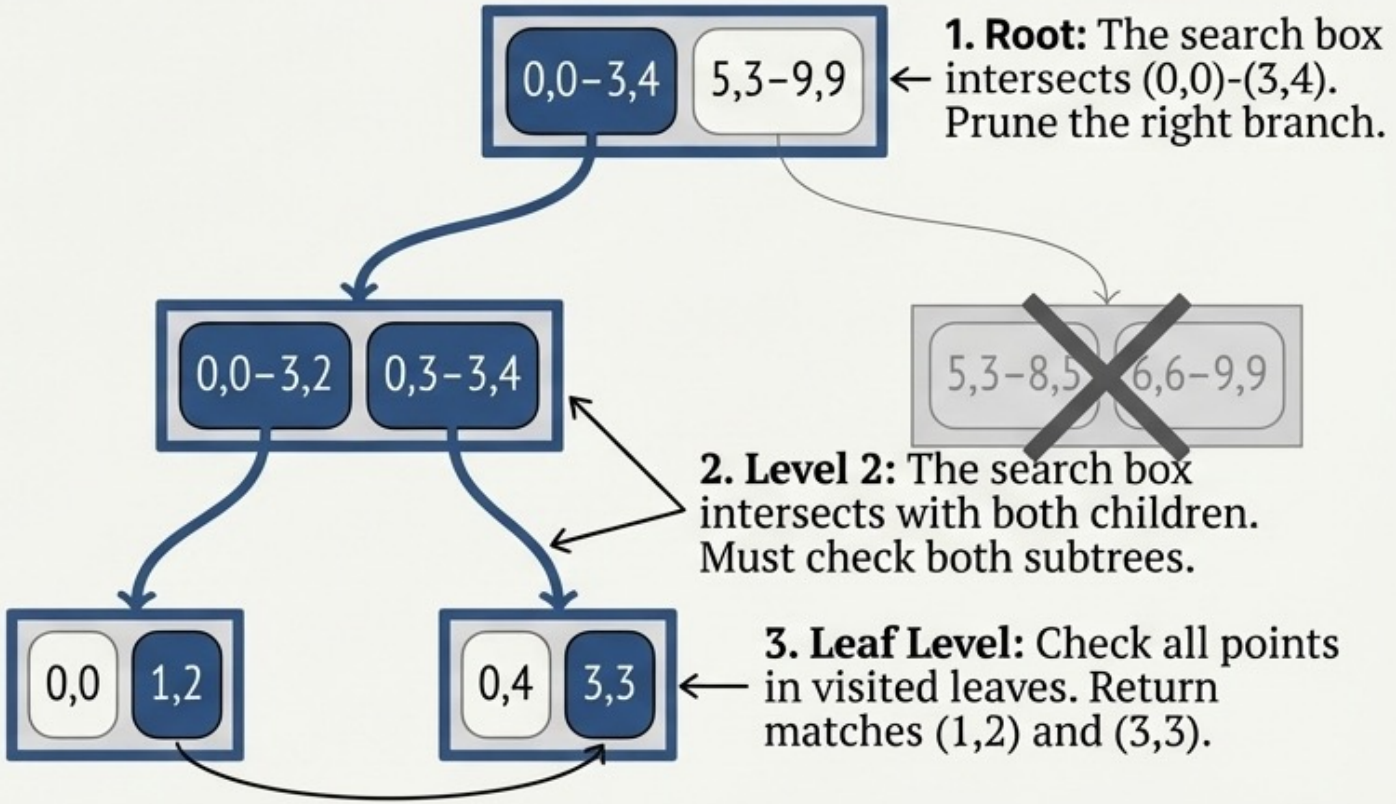
NotebookLM

How Search Works: Finding Points Inside a Region

```
WHERE coordinates <@ 'search_area'::box
```

The 'consistency' function for this query (gist_point_consistent) checks if the search area *intersects* with the bounding box of an index entry.





NotebookLM

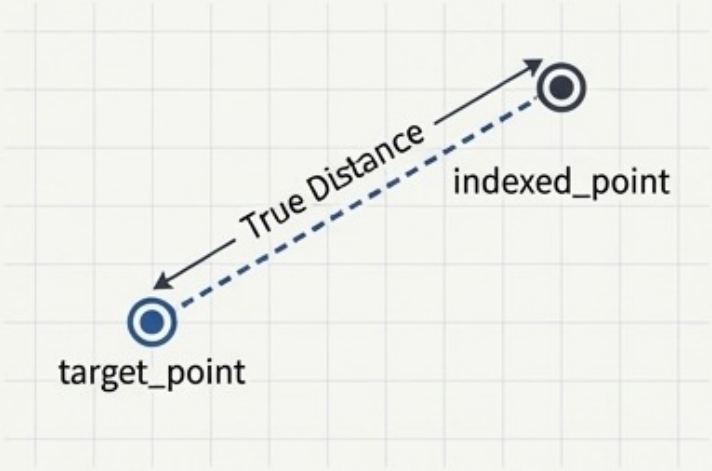
Beyond Containment: Finding the Nearest Neighbors (k-NN)

```
ORDER BY coordinates <-> 'target_point'::point LIMIT k
```

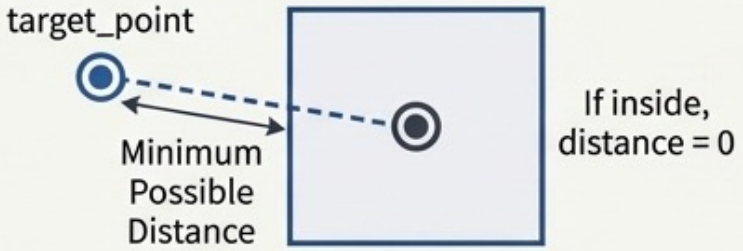
This is powered by an “ordering operator” (`<->`) and a new GiST support function: `` `distance` ``.

The `distance` Function

For Leaf Nodes (Points): Returns the true Euclidean distance from the indexed point to the target point.



For Internal Nodes (Bounding Boxes): This is the crucial part. It returns the *minimum possible distance* from the target point to *any point within the box*. If the point is inside the box, the distance is 0. This is an optimistic, lower-bound estimate.

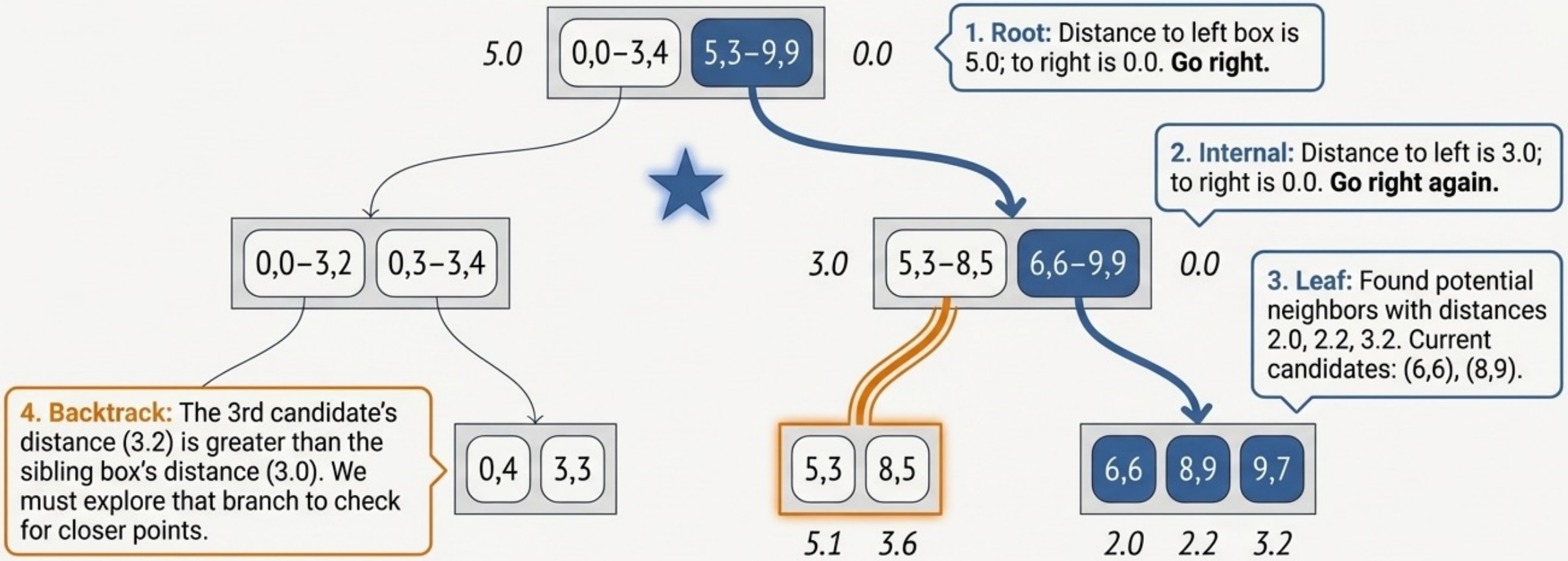


The search traverses the tree, always visiting the child node with the smallest distance first. It maintains a priority queue of branches to visit. This ensures it finds the closest points as quickly as possible.

NotebookLM

Visualizing the k-NN Search Algorithm

Goal: Find the 3 nearest neighbors to the point (6,8) (represented by the star icon).



The underestimated distance for internal nodes ensures correctness but can lead to exploring more branches, demonstrating a performance trade-off.

NotebookLM

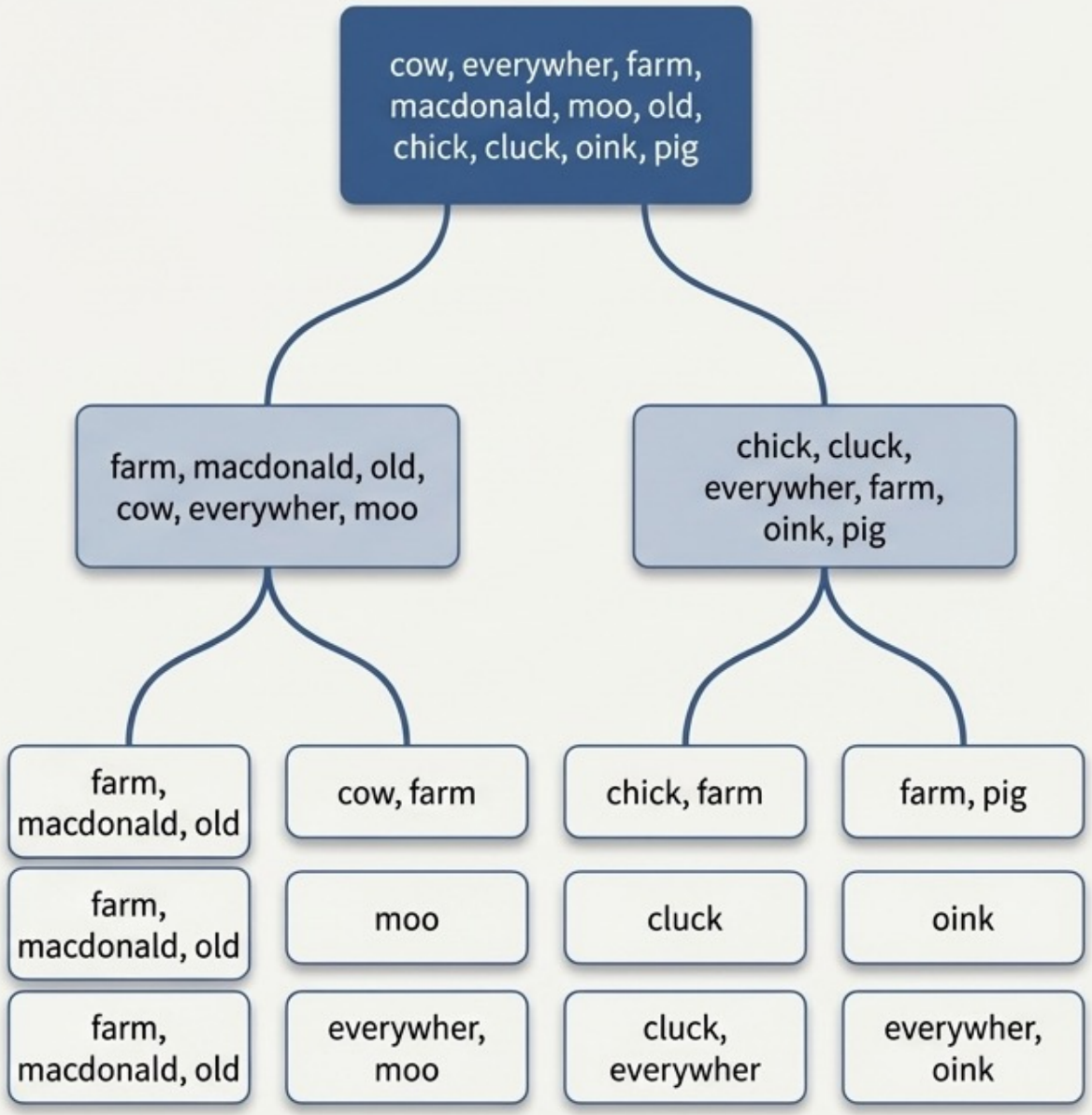
Case Study 2: RD-trees for Full-Text Search

The Problem

How do you index documents to find ones containing specific words or phrases (e.g., `tsvector @@ tsquery`)? Bounding boxes are meaningless for sets of words.

The RD-tree Solution

- 'RD" stands for Russian Doll. It's a GiST implementation that uses bounding *sets* instead of bounding boxes.
- **Predicate:** "The set of words in this document is a subset of this bounding set."
- **Hierarchy:** An internal node's predicate is the *union* of all unique words (lexemes) found in its child documents.
- **'consistency' function:** Checks if the lexemes in the search query are present in the node's bounding set.

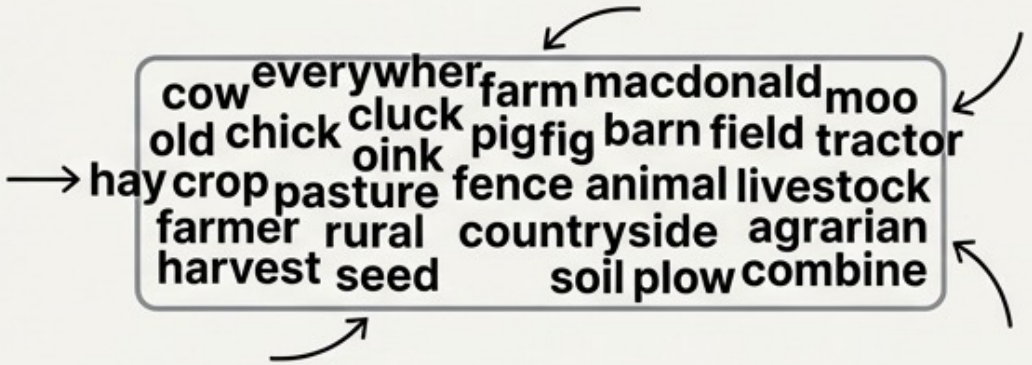


NotebookLM

The Challenge of Large Sets and the Signature Tree Solution

The Problem

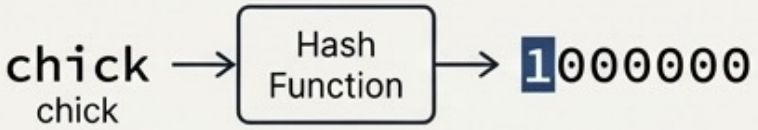
Storing a union of all unique words for high-level nodes is inefficient. The sets can grow larger than a single index page.



The Solution: Signature Trees

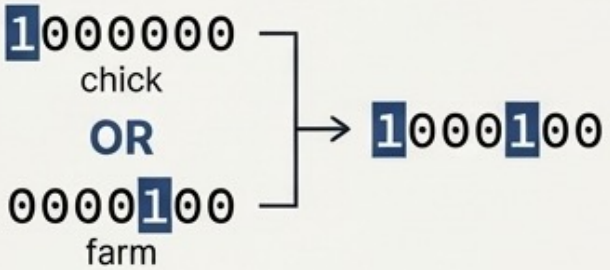
Instead of storing the words, we store a fixed-size bitmask signature (similar to a Bloom Filter).

1. Hashing



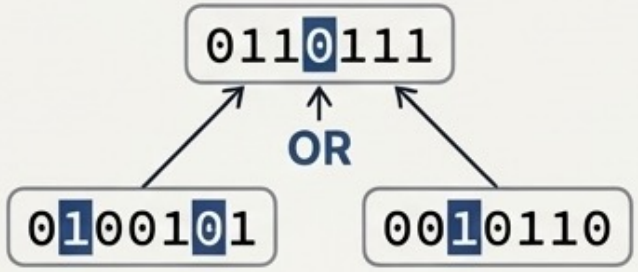
Each unique lexeme is hashed to a position in a bit string, setting that bit to '1'.

2. Document Signature



A document's signature is the bitwise `OR` of all its lexemes' signatures.

3. Tree Hierarchy



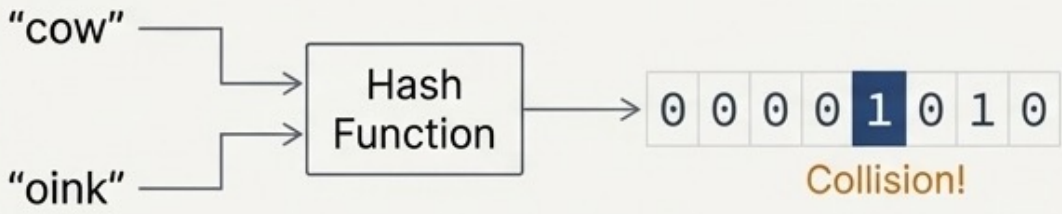
An internal node's signature is the bitwise `OR' of all its children's signatures.

NotebookLM

The Trade-off: Compactness vs. False Positives

The Catch

Search Logic: `consistency` function checks if `(query_signature & node_signature) == query_signature`.



Since the signature has a fixed size, different words can hash to the same bit.

The index returns rows that match the signature but not the actual query. These **false positives** must be filtered out by a “recheck”. However, signature trees **never produce false negatives**.

Real-World Example

Query: A perfectly typeset code block shows:

```
EXPLAIN (analyze, costs off, timing off, summary off)
    SELECT *
    FROM mail_messages
    WHERE tsv @@ to_tsquery('magic & value');
```

```
Index Scan using mail_gist_idx on mail_messages (actual rows=898 loops=1)
  Index Cond: (tsv @@ to_tsquery('magic & value'::text))
  Rows Removed by Index Recheck: 7859
```

The index returned 8,757 rows (898 + 7859), but 90% were **false positives** that had to be filtered out.

NotebookLM

Tuning for Performance: Managing the False Positive Rate

The Knob

You can control the size of the signature using the `siglen` operator class parameter when creating the index.

The Trade-off

Larger `siglen`	**Smaller `siglen` (default: 124 bytes)**
• Fewer hash collisions. • Fewer false positives & less recheck work. • Better search performance. • Larger index size on disk.	• More compact index. • Higher chance of false positives. • Potentially slower queries on large datasets.

Example

Increasing `siglen` from the default to 248 for the email archive.

Index Size

127 MB → 139 MB

A modest increase

Rows Removed by Index Recheck

7,859 → 2,060

A 74% reduction in false positives

Tuning `siglen` is a critical optimization for GiST-based full-text search.

NotebookLM

A Unique Power: Enforcing Complex Rules with Exclusion Constraints

An exclusion constraint ensures that for any two rows in a table, a specified operator will not return `TRUE`. `TRUE`. It's a generalization of `UNIQUE` constraints, supported by GiST with operators like `&&` (overlaps).

Simple Example

Prevent airports from being located too close to each other.

```
ADD EXCLUDE USING gist
  (circle(coordinates, 0.2)
  WITH &&);
```



Advanced Multi-Column Example

Allow airports to be close *only* if they are in the same city.

Requires the `btree_gist` extension to use a standard operator (`!=`) on a text column within a GiST index.

```
ADD EXCLUDE USING gist
  (circle(coordinates, 0.2) WITH &&,
  (city->>'en') WITH !=);
```



NotebookLM

GiST Properties at a Glance

Access Method Properties

`can_multi_col` Can index multiple columns	Yes
`can_exclude` Supports exclusion constraints	Yes
`can_include` Supports `INCLUDE` columns for index-only scans, opclass permitting	Yes
`can_unique` Cannot enforce `UNIQUE` constraints directly	No
`can_order` Does not produce pre-sorted output like a B-tree	No

Index Scan Properties

`index_scan`	Yes
`bitmap_scan`	Yes
`backward_scan`	No

Column Level Properties (Operator Class Dependent!)

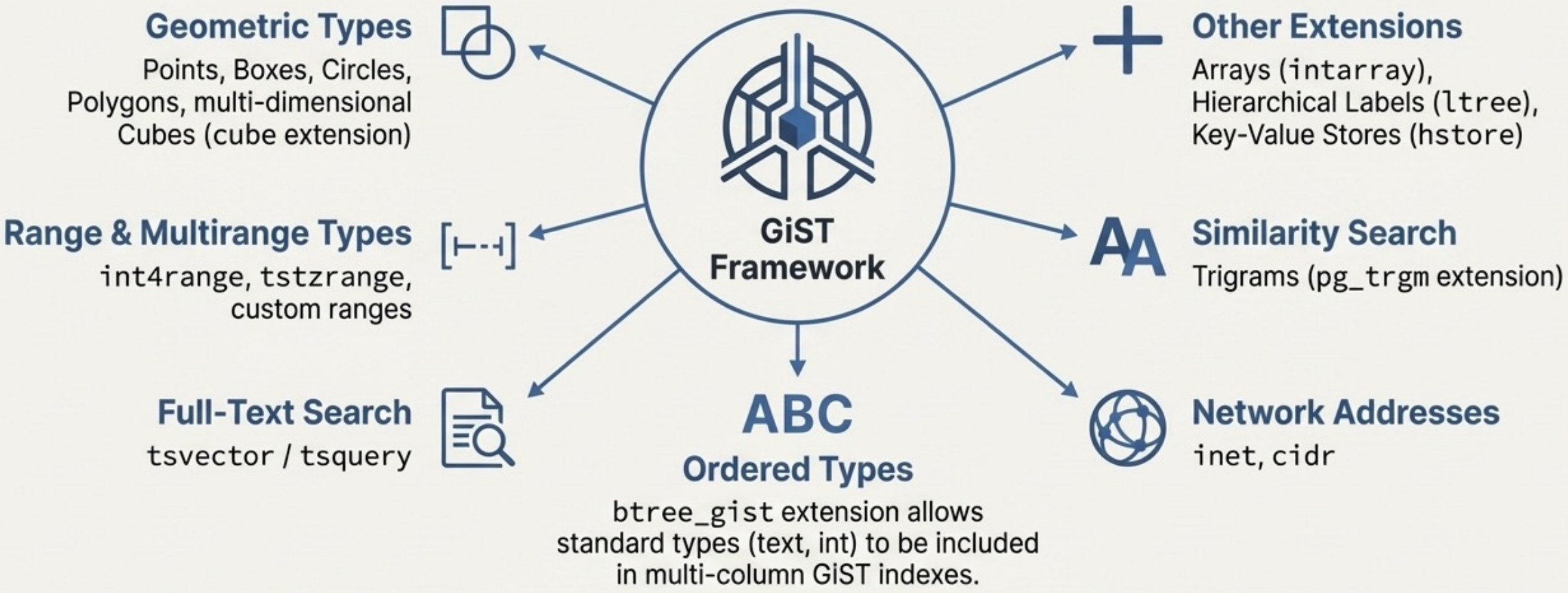
`search_nulls` But not always efficient	Yes
`returnable` (for Index-Only Scans) Depends on Opclass Yes for R-trees, No for Signature Trees	
`distance_orderable` (for k-NN) Depends on Opclass Yes for `point_ops` No for `tsvector_ops` or range types	

Key takeaway: Many critical GiST behaviors are determined not by GiST itself, but by the specific operator class you choose.

NotebookLM

A Framework for a Universe of Data Types

GiST's power comes from its extensibility. By implementing a handful of support functions, it can be adapted to provide high-performance indexing for a wide variety of data.



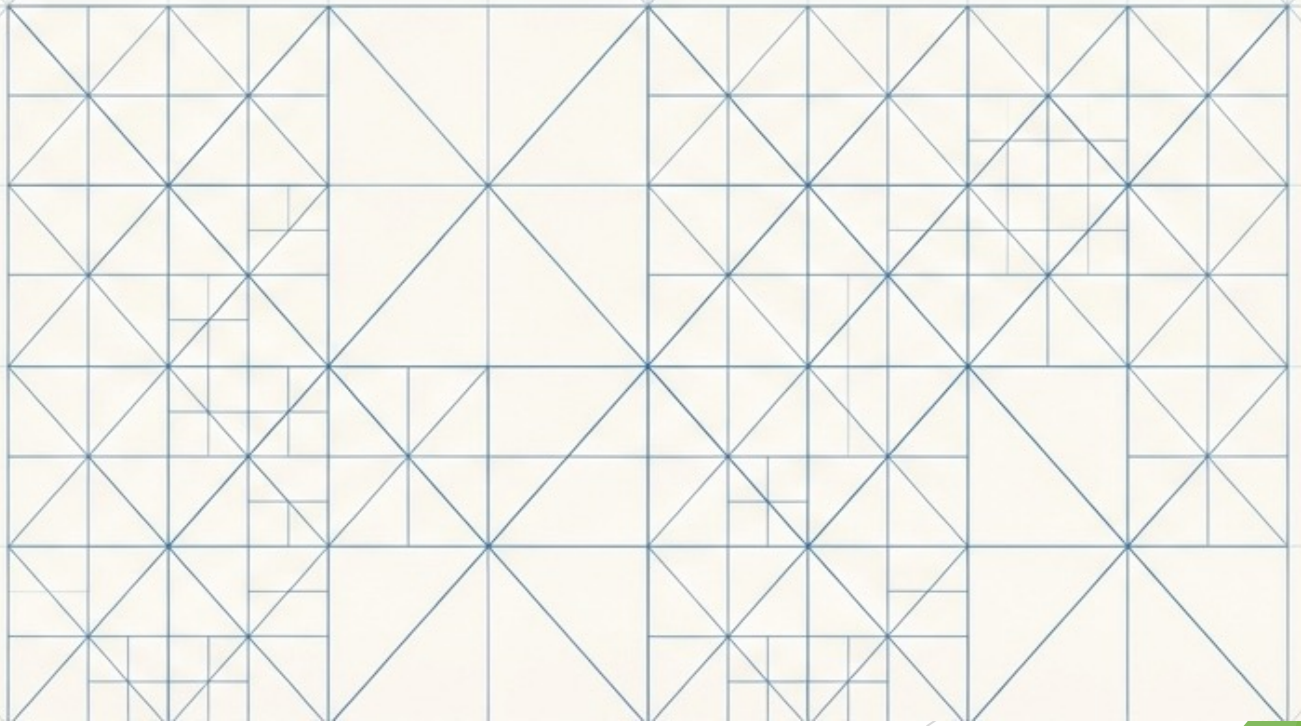
When you encounter complex data that doesn't fit a B-tree, GiST provides the framework to build a specialized, high-performance solution.

NotebookLM

27. SP-GiST

SP-GiST: Building Specialized Tree Structures Inside PostgreSQL

An exploration of the Space-Partitioned Generalized Search Tree, a framework for implementing structures like Quadtrees and Radix trees.



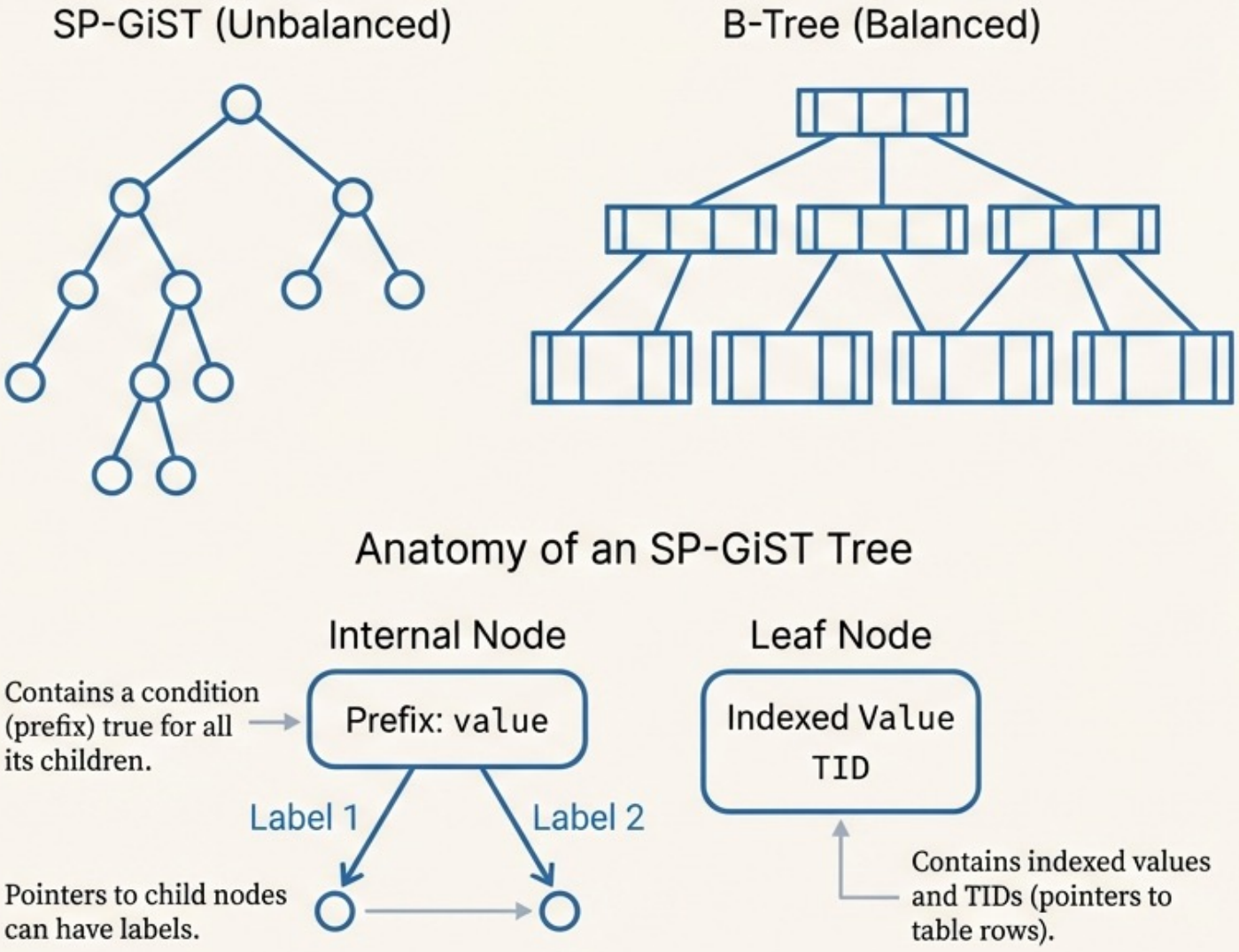
NotebookLM

The Core Idea: Recursive Space Partitioning

SP-GiST stands for **Space-Partitioned** Generalized Search Tree. It works by recursively dividing a search space into non-overlapping regions.

How it's different:

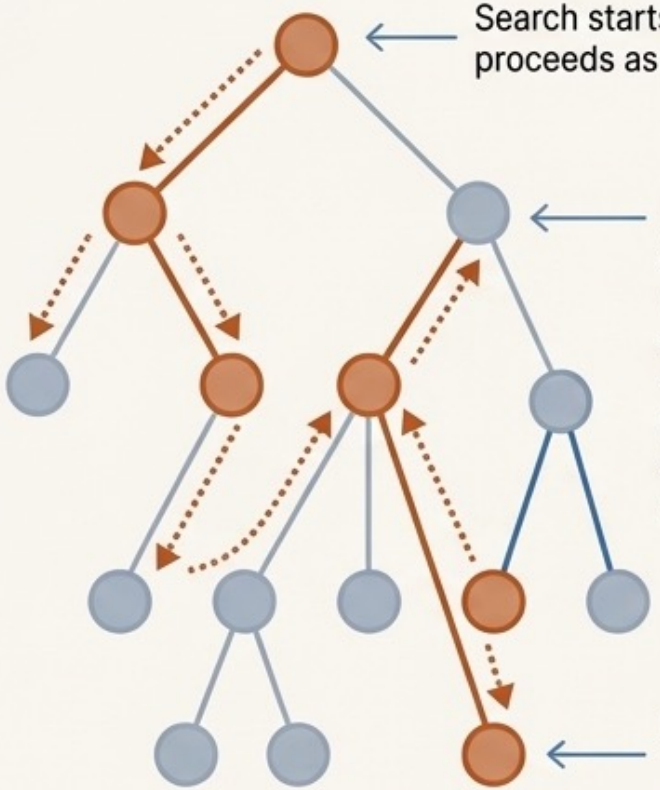
- This creates **unbalanced trees,** which can have varying branch depths based on data density.
- This is unlike B-trees or even GiST, which create balanced trees where each node typically occupies a full disk page.
- SP-GiST is a framework for implementing well-known structures like **Quadtrees, k-D trees, and Radix trees (tries).**



NotebookLM

How SP-GiST Navigates and Grows

Search: A Depth-First Traversal

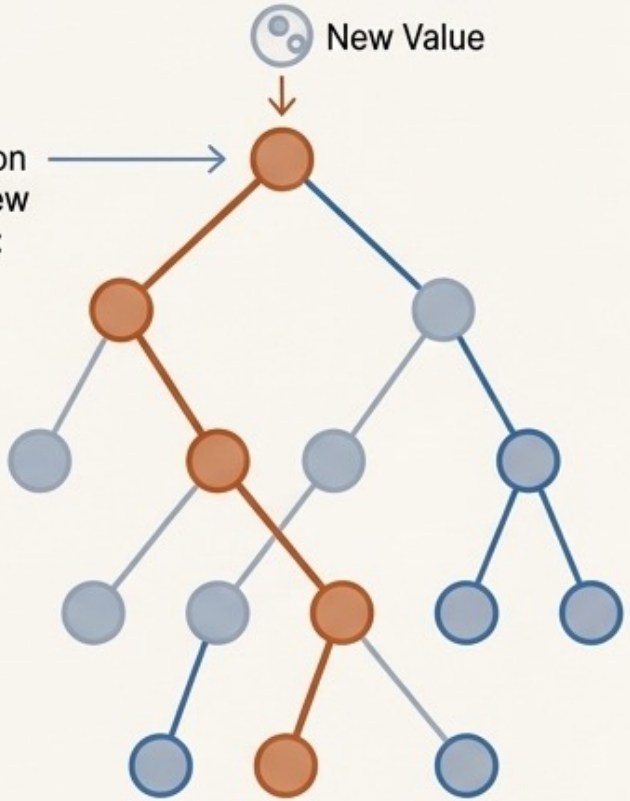


Search starts at the root and proceeds as a depth-first search.

A `consistent' function evaluates internal nodes (based on their prefixes and labels) to determine which child branches are consistent with the search query and worth traversing.

For leaf nodes, the `consistent` function checks if the actual indexed values satisfy the search condition.

Insertion: A Guided Placement



The `choose' support function determines the path for a new value. It decides whether to:

1. Send the value down to an existing child node.
2. Create a new child node for the value.
3. Split the current node if the new value doesn't match its prefix.

If a leaf page becomes full, the `picksplit` function decides which nodes to move to a new page.

NotebookLM

Deep Dive 1: Quadtrees for 2D Geographic Points

The Concept: Quadtrees are used to index two-dimensional points. The plane is recursively partitioned into four regions (quadrants) around a selected 'center point'.

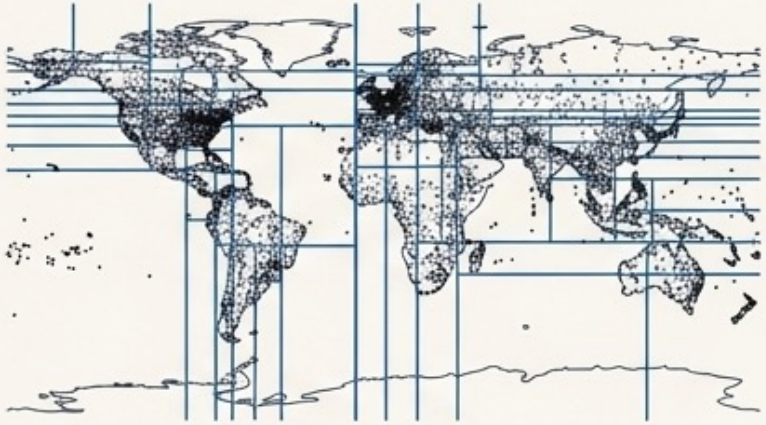
The following diagrams show this partitioning applied to a world map of airports. The density of points dictates the depth of partitioning in each region.



Step 1: The Initial Plane



Step 2: First Partition



Step 3: Recursive Partitioning

The index for this visualization was created with:

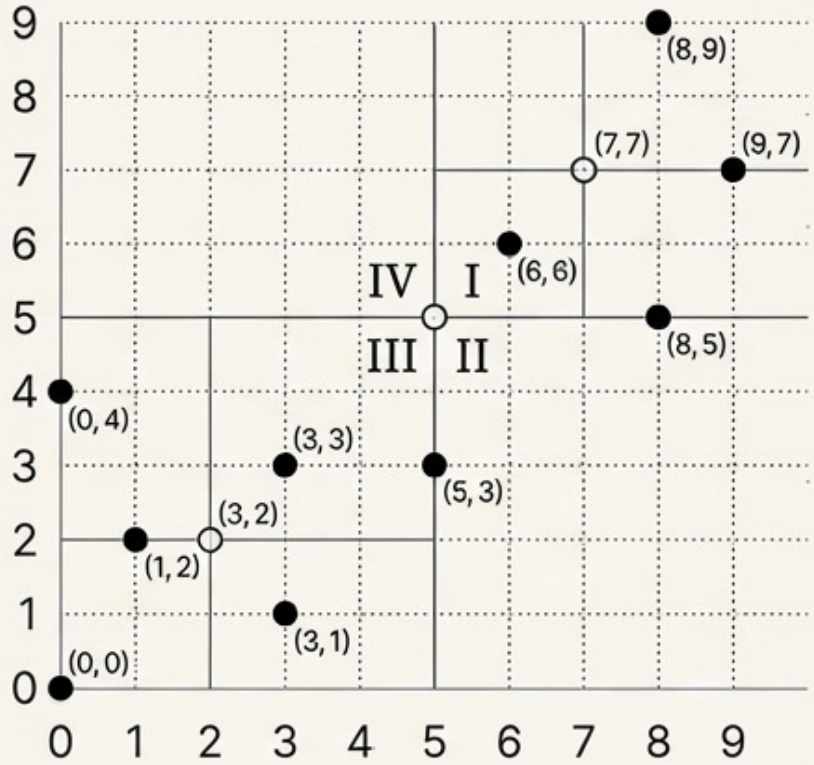
```
CREATE INDEX airports_quad_idx ON airports_big USING spgist(coordinates) WITH (fillfactor = 10);
```

(Note: A low `fillfactor` was used here to force a deeper, more illustrative tree.)

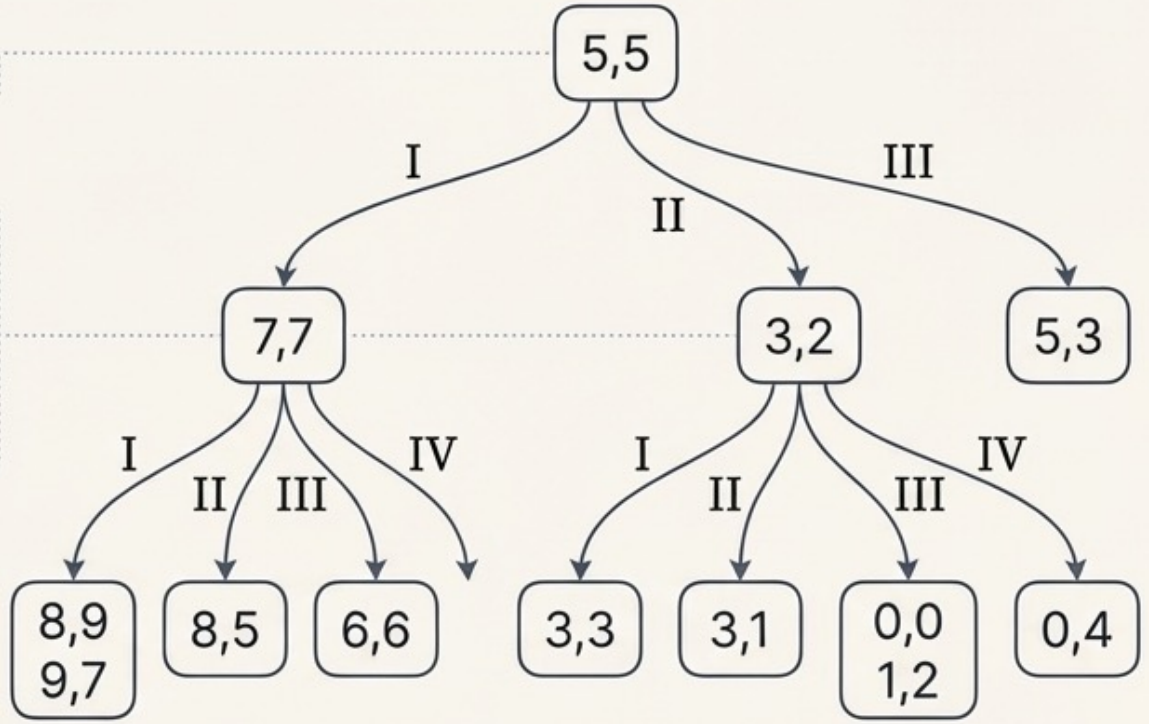
NotebookLM

From a Partitioned Plane to a Search Tree

Partitioned Plane



Quadtree Structure

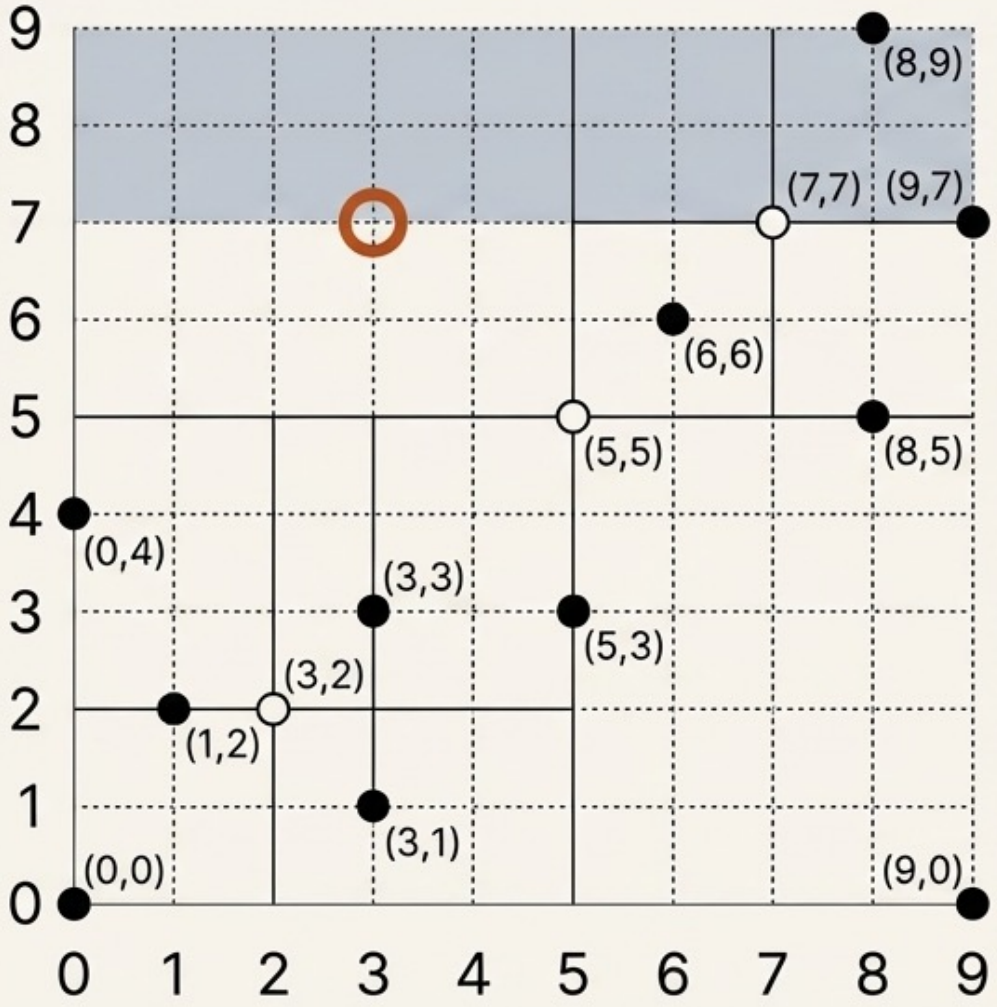


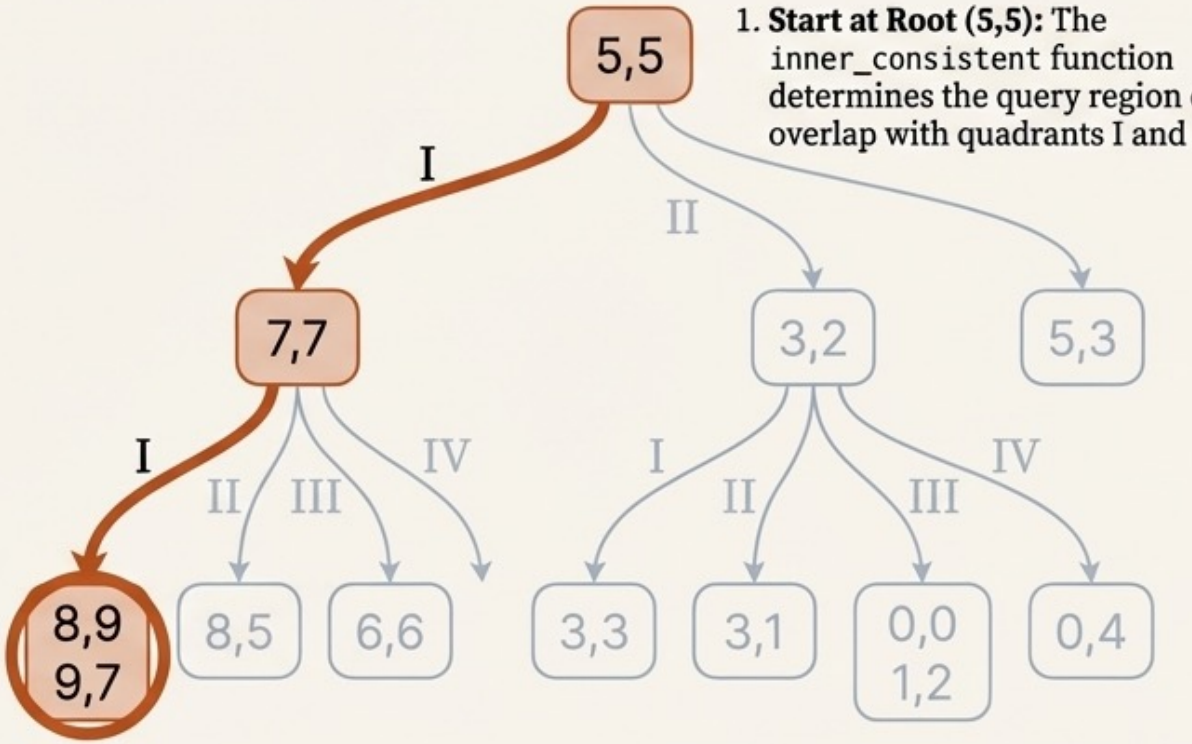
The geometric partitioning of the 2D space directly **translates** into a **hierarchical tree** structure that the database can traverse.

NotebookLM

Walking the Tree: How a Quadtree Search Works

Query: Find all points located *above* the point (3,7).





1. **Start at Root (5,5):** The `inner_consistent` function determines the query region could overlap with quadrants I and IV.

2. Descend to Quadrant I (Node 7,7): The function again checks for overlap overlap. The query region is consistent with this node's quadrants I and IV.

4. Return to Root, Check Quadrant IV: The node recuses coıts (8,9) matches. (Note: Quadrant IV is empty).

3. Check Leaf (Quadrant I of 7,7): The `leaf_consistent` function compares points (8,9) and (9,7) against the query condition > (3,7). The node corresponding to the root's quadrant IV is empty. The search concludes.

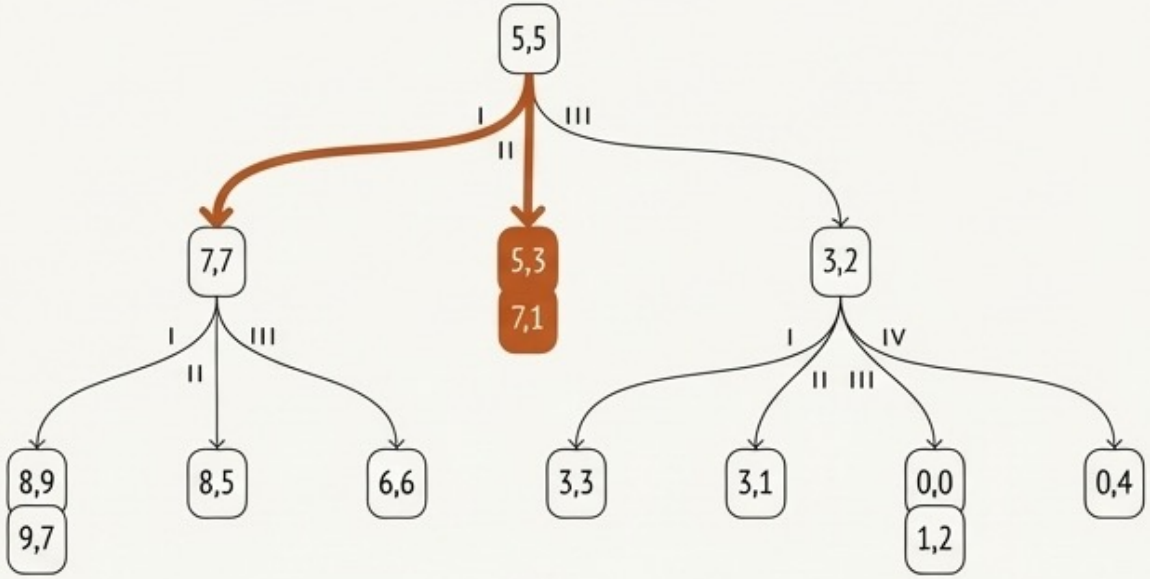
NotebookLM

Growing the Tree: Insertion and Node Splitting

Case 1: Simple Insertion

Action: Insert point (7,1).

Path: The choose function traverses the tree: at root (5,5), (7,1) falls into Quadrant II. The point is added to the leaf node (5,3).

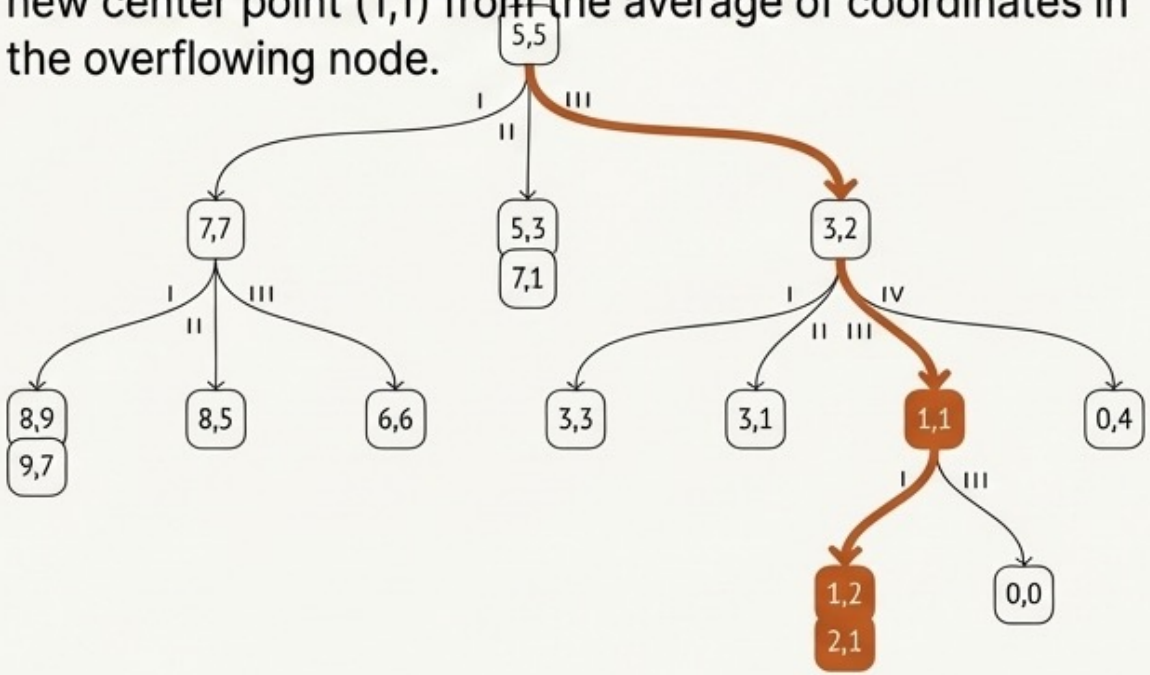


Case 2: Insertion Causing a Split

Action: Insert point (2,1). This causes the leaf page for its target quadrant to overflow.

Path: The choose function directs (2,1) towards the leaf list containing (0,0), (1,2), (3,1).

Split: The picksplit function is invoked. It calculates a new center point (1,1) from the average of coordinates in the overflowing node.

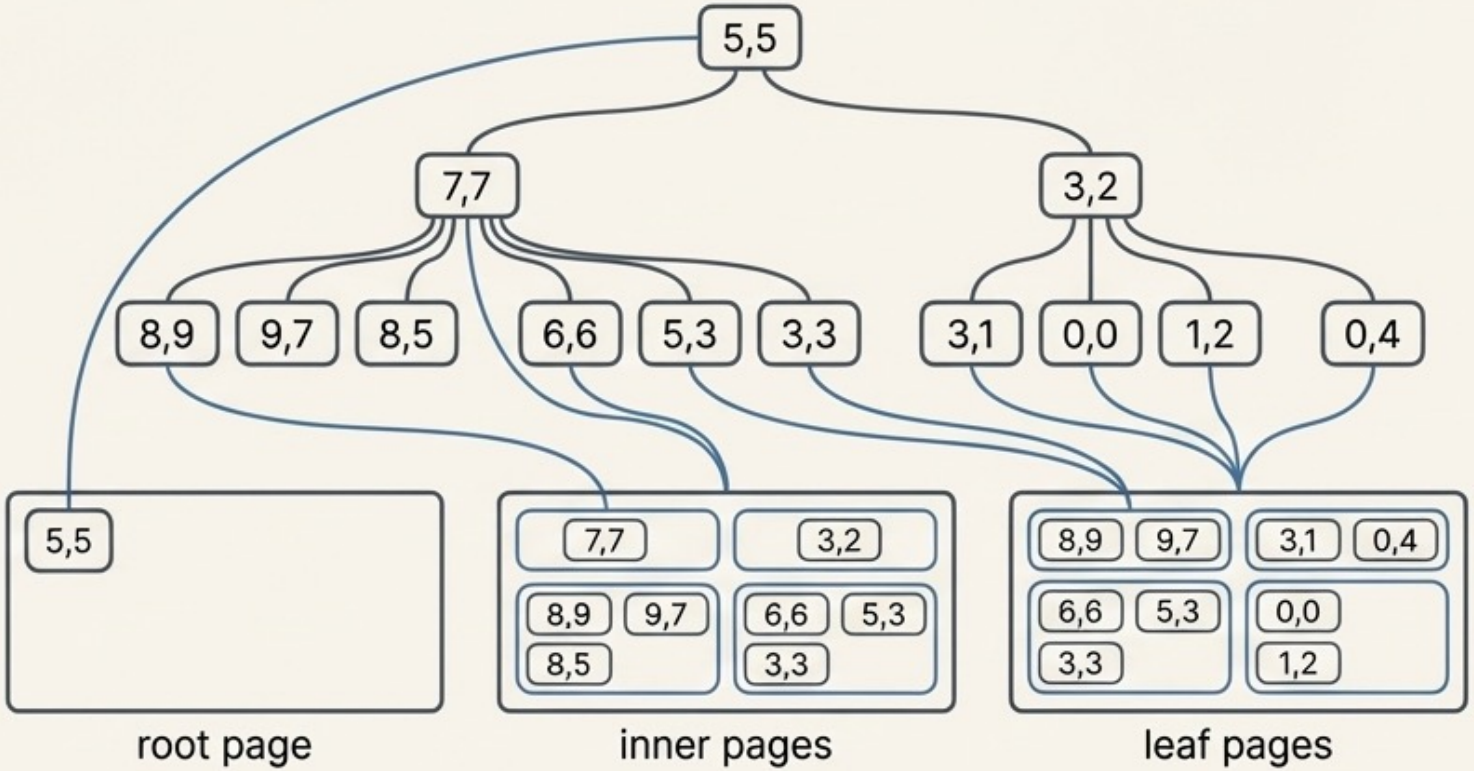


NotebookLM

Under the Hood: Page Structure & Operator Classes

Physical Page Layout

- There is no 1:1 mapping between tree nodes and disk pages. Multiple small nodes are packed onto a single page to minimize I/O.
- **Internal Pages** store internal nodes (prefix + child pointers).
- **Leaf Pages** store leaf nodes (value + TID). All leaf nodes for a given parent are linked together on a single page.
- **Root node is always on page 1.** Page 0 is a meta page.



The Operator Class: `quad_point_ops`

The specific logic for Quadtrees is defined by an operator class. It provides the required support functions:

- `config`: Reports basic info to the access method.
- `choose`: Selects a node for insertion.
- `picksplit`: Distributes nodes after a page split.
- `picksplit`: Distributes nodes after a page split.
- `inner_consistent`: Checks internal nodes during a scan.
- `leaf_consistent`: Checks leaf nodes during a scan.

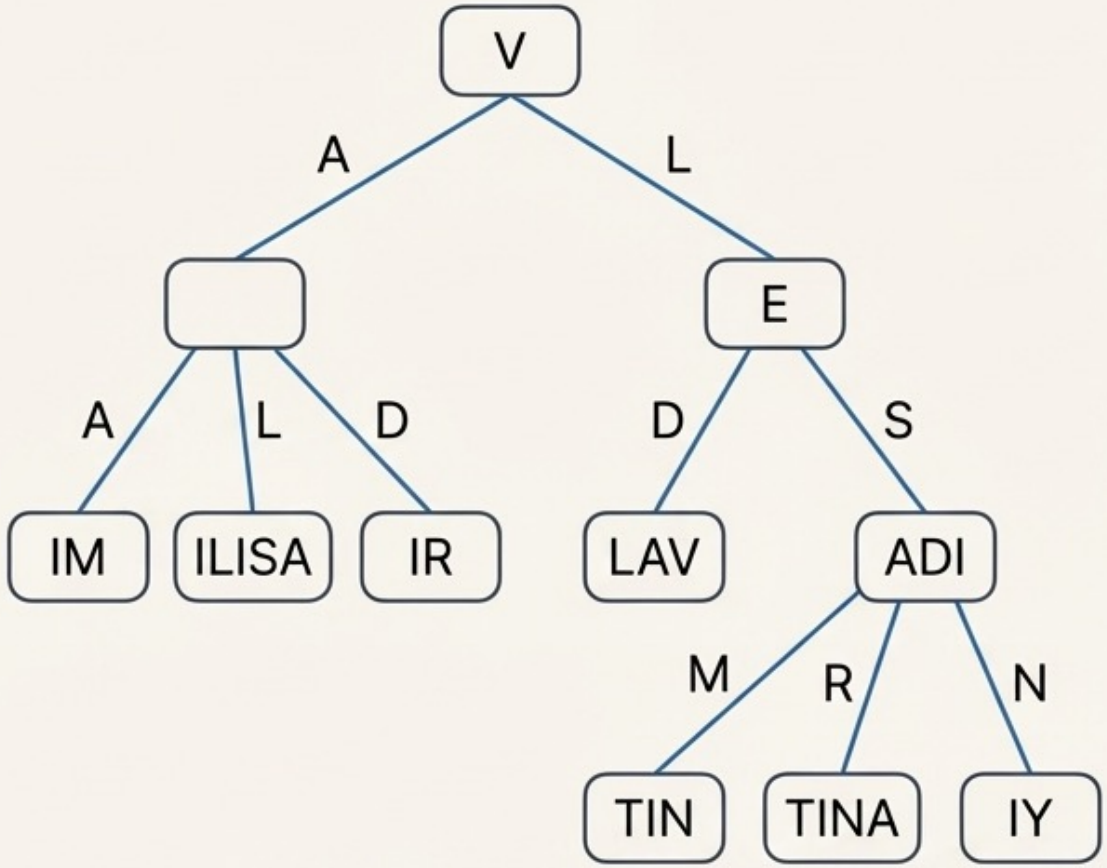
NotebookLM

Deep Dive 2: Radix Trees for Efficient Text Indexing

The Concept: The `text_ops` operator class implements a **Radix Tree** (or trie). This structure excels at prefix-based searches.

How it Works:

- **Internal Nodes** store a common **prefix** shared by all child values.
- **Pointers** to child nodes are **labeled** with the next character (or byte) that follows the prefix.
- **Leaf Nodes** store only the remaining suffix of the value.



Key Feature:

To reconstruct a full value, you concatenate all prefixes and labels on the path from the root to the leaf. This can result in a much more compact compact index than a B-tree, which stores full values.

NotebookLM

Radix Trees in Action: Prefix and Range Searches

Use Case: Prefix Search

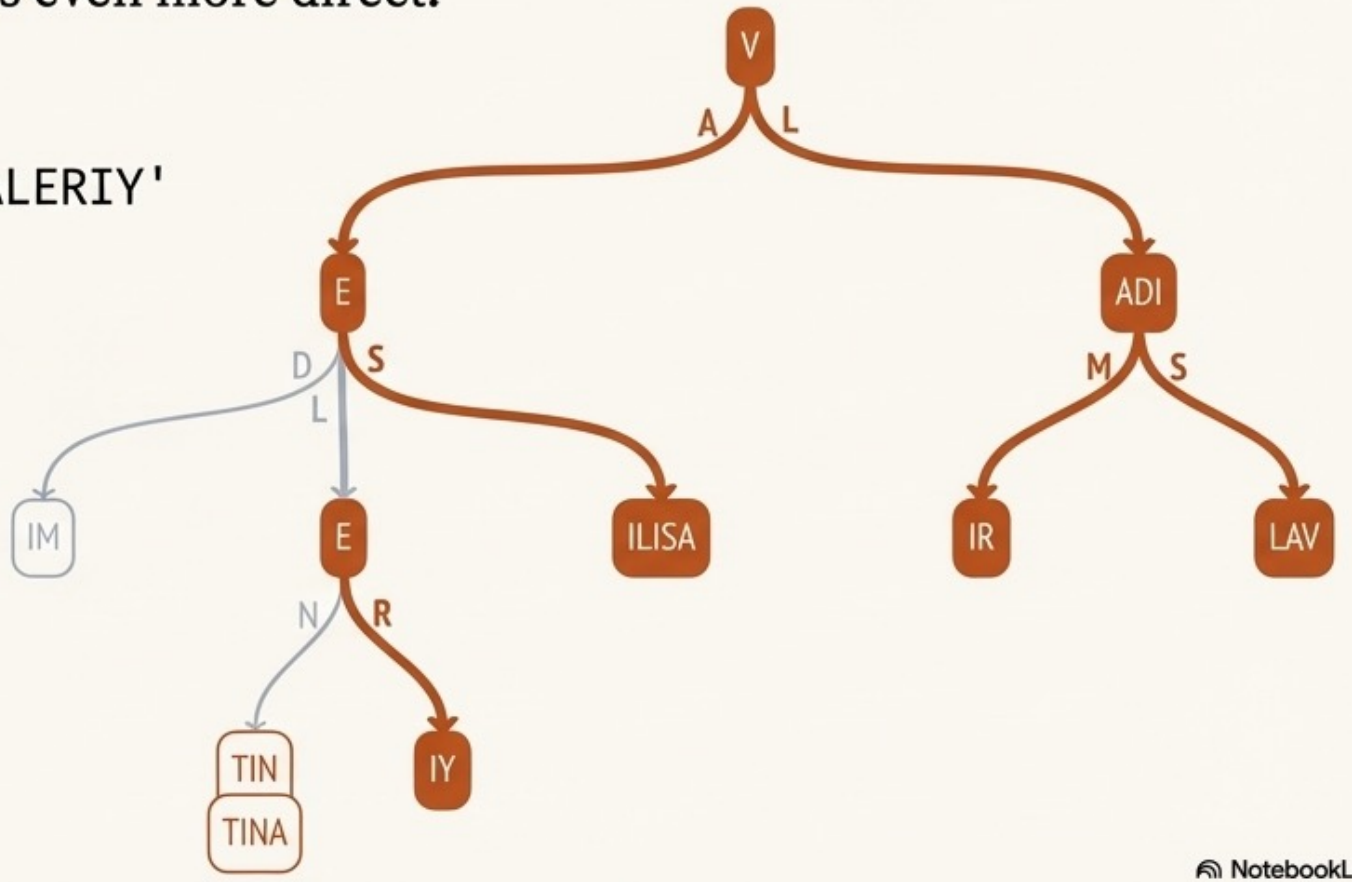
- Radix trees are highly efficient for prefix searches.
- The `LIKE 'IVAN%'` condition is translated by the planner into a range scan: `passenger_name ~>= 'IVAN''` and `passenger_name ~<~ 'IVAO'`.
- The dedicated `^@` (`starts_with`) operator is even more direct.

Walkthrough: Range Scan

```
SELECT * FROM names WHERE name ~>=~ 'VALERIY'
      AND name ~-<~ 'VLADISLAV';
```

The `inner_consistent` function traverses the tree, concatenating prefixes and labels (e.g., `V + L → VL`). It compares the reconstructed prefix at each step with the query bounds.

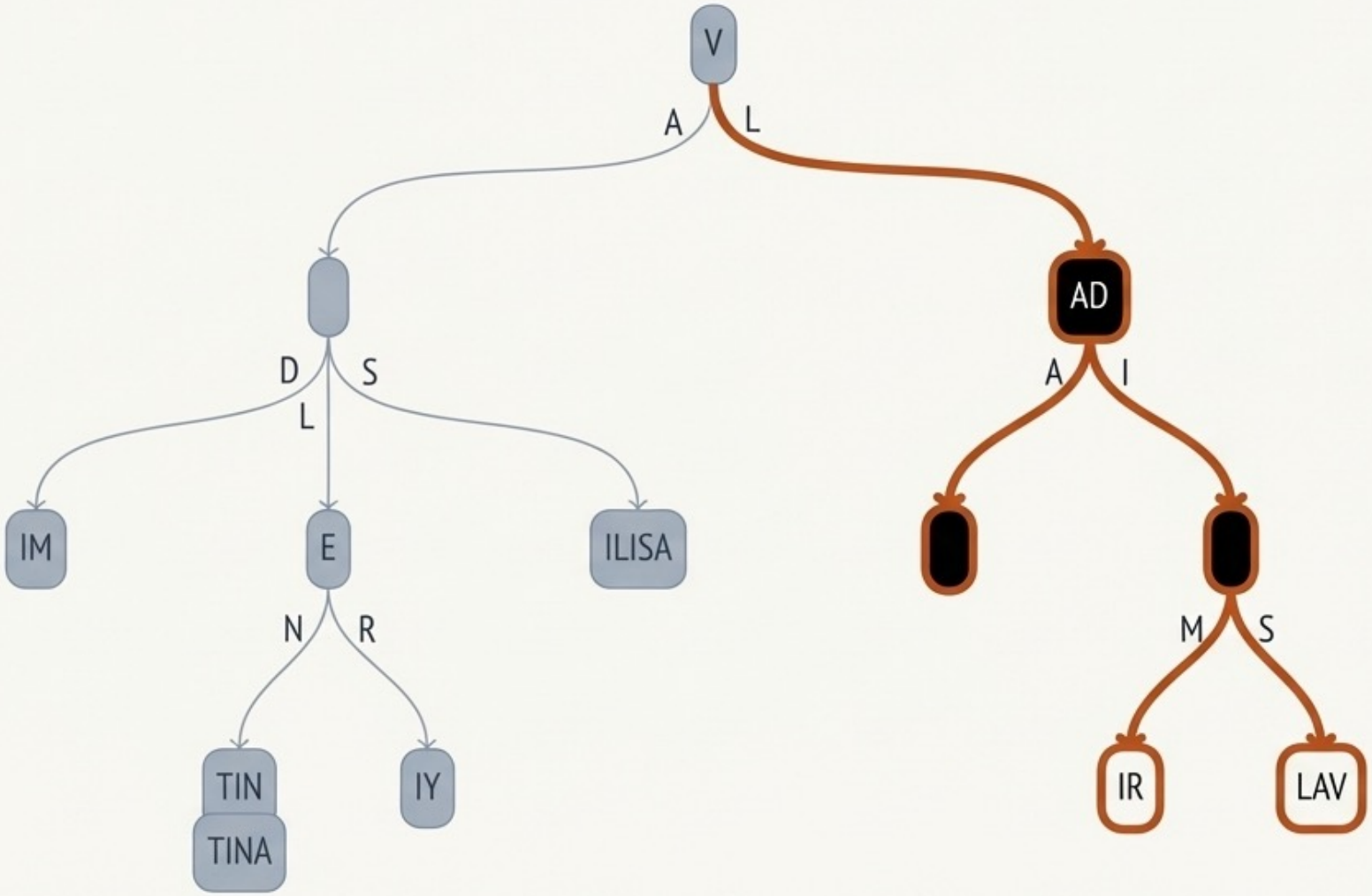
Branches that fall outside the range (e.g., `VAD` because `VAD < 'VAL'`) are pruned, dramatically reducing the search space.



NotebookLM

The Challenge of Radix Tree Insertion

The Problem: Unlike points in a quadrant, a new string may not share an existing prefix. This requires restructuring the tree.



1. The `choose` function traverses from the root `V` down the `L` branch.
2. The next node has the prefix `ADI`.
3. The remainder of the new string, `ADA`, does not match `ADI`.
4. **Action: Node Split.** The common prefix `AD` is identified. A new internal node with prefix `AD` is created.
5. The old node's remaining prefix `I` and the new string's remaining character `A` become labels for the new child pointers.

NotebookLM

The SP-GiST Property Checklist

Access Method Properties (`pg_indexam_has_property`)

can_order	false
can_unique	false
can_multi_col	false
can_exclude	true
can_include	true

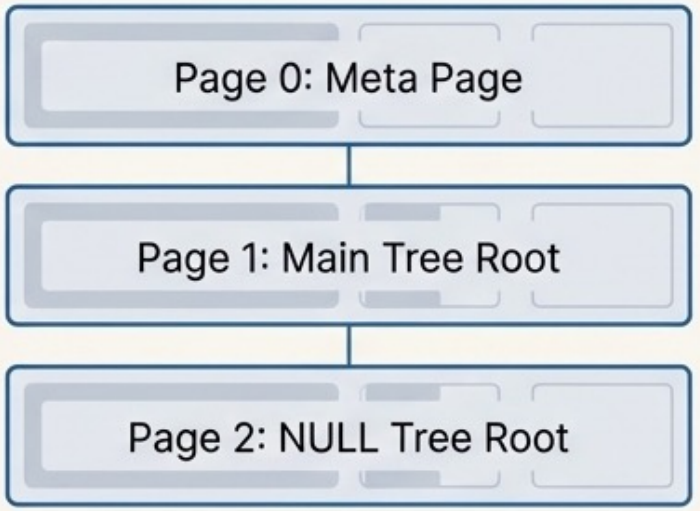
Index-Level Properties (`pg_index_has_property`)

clusterable	false
bitmap_scan	true

Special Handling for NULLs

NULL values are not stored in the main tree.

A separate SP-GiST tree for NULL entries is created, with its root at page 2 of the index file.



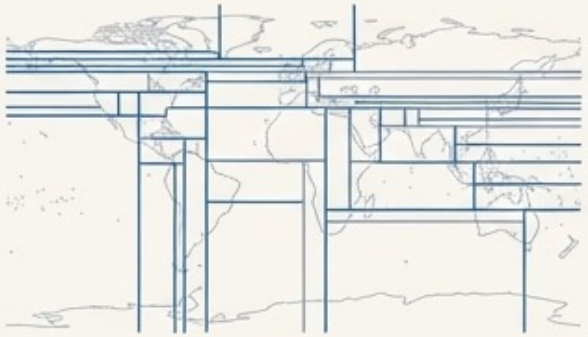
NotebookLM

A Framework for Diverse Data Structures

SP-GiST is a foundation for many index types:

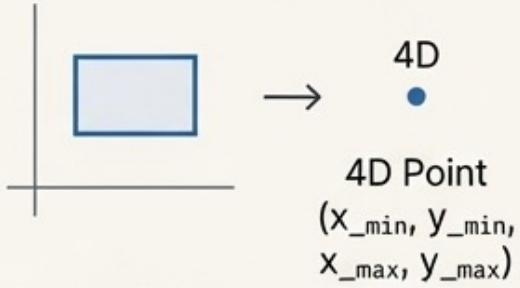
k-D Trees (kd_point_ops)

An alternative for 2D points that partitions space by alternating between horizontal and vertical splits, always creating two sub-regions instead of four.



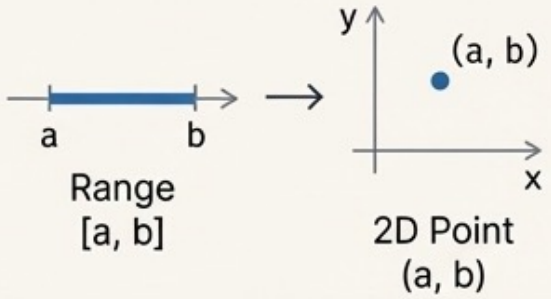
Geometric Types (box_ops)

Implements a Quadtree for rectangles by representing them as points in 4D space.



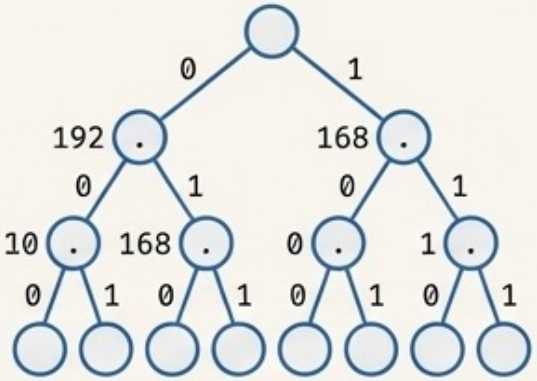
Range Types (range_ops)

Implements a Quadtree for ranges, where a range [a, b] is treated as a 2D point (a, b).



Network Addresses (inet_ops)

Implements a Radix Tree for IP addresses, enabling efficient prefix-based lookups on subnets.



Key Takeaway: SP-GiST provides the core engine for concurrency, logging, and tree traversal, allowing developers to implement complex, specialized, and efficient non-balanced tree indexes by simply defining the partitioning logic in an operator class.

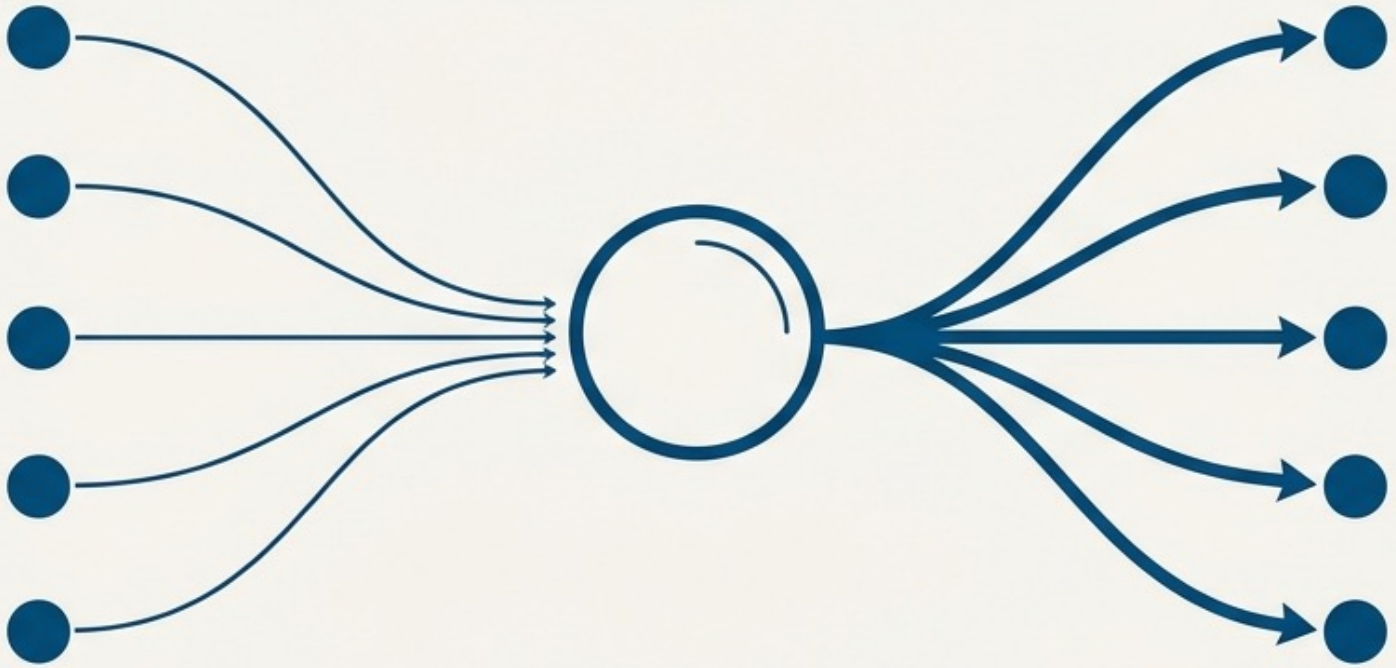
NotebookLM

28. GIN

GIN: The Generalized Inverted Index

Unlocking High-Performance Search for Complex Data in PostgreSQL

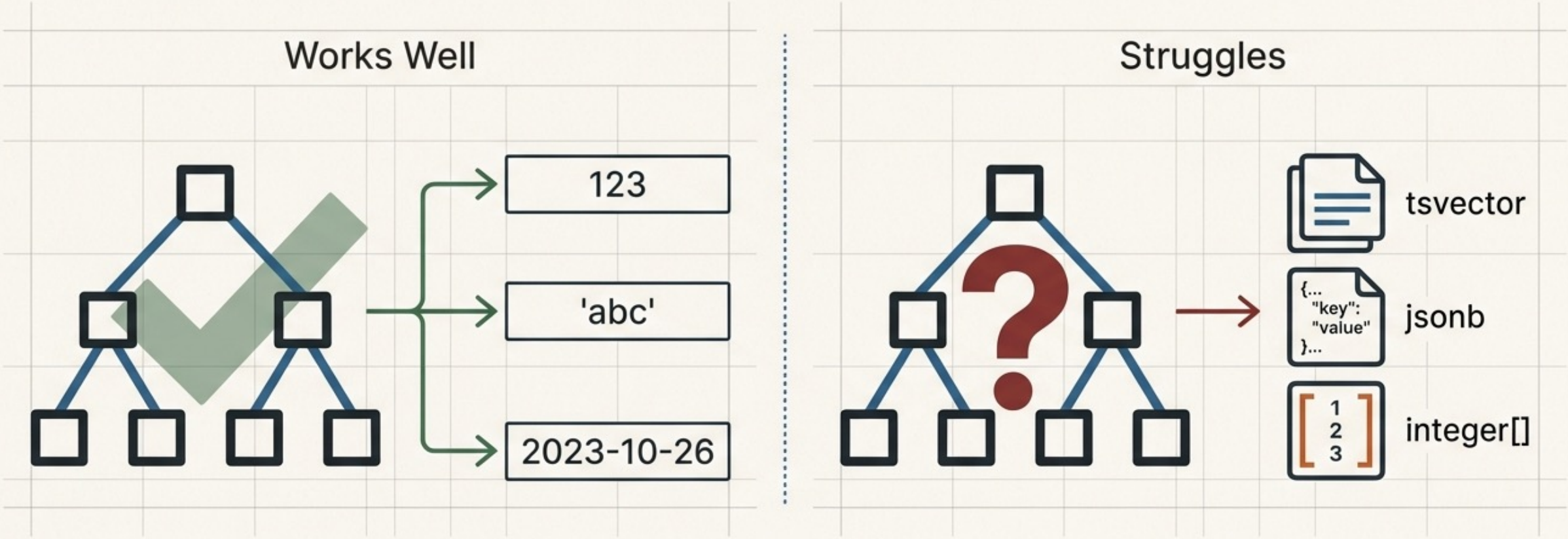
The name GIN officially stands for Generalized Inverted Index. Its approach is designed for indexing non-atomic values composed of separate elements, such as the lexemes within a document or the elements within an array.



NotebookLM

Standard Indexes Struggle with Composite Data

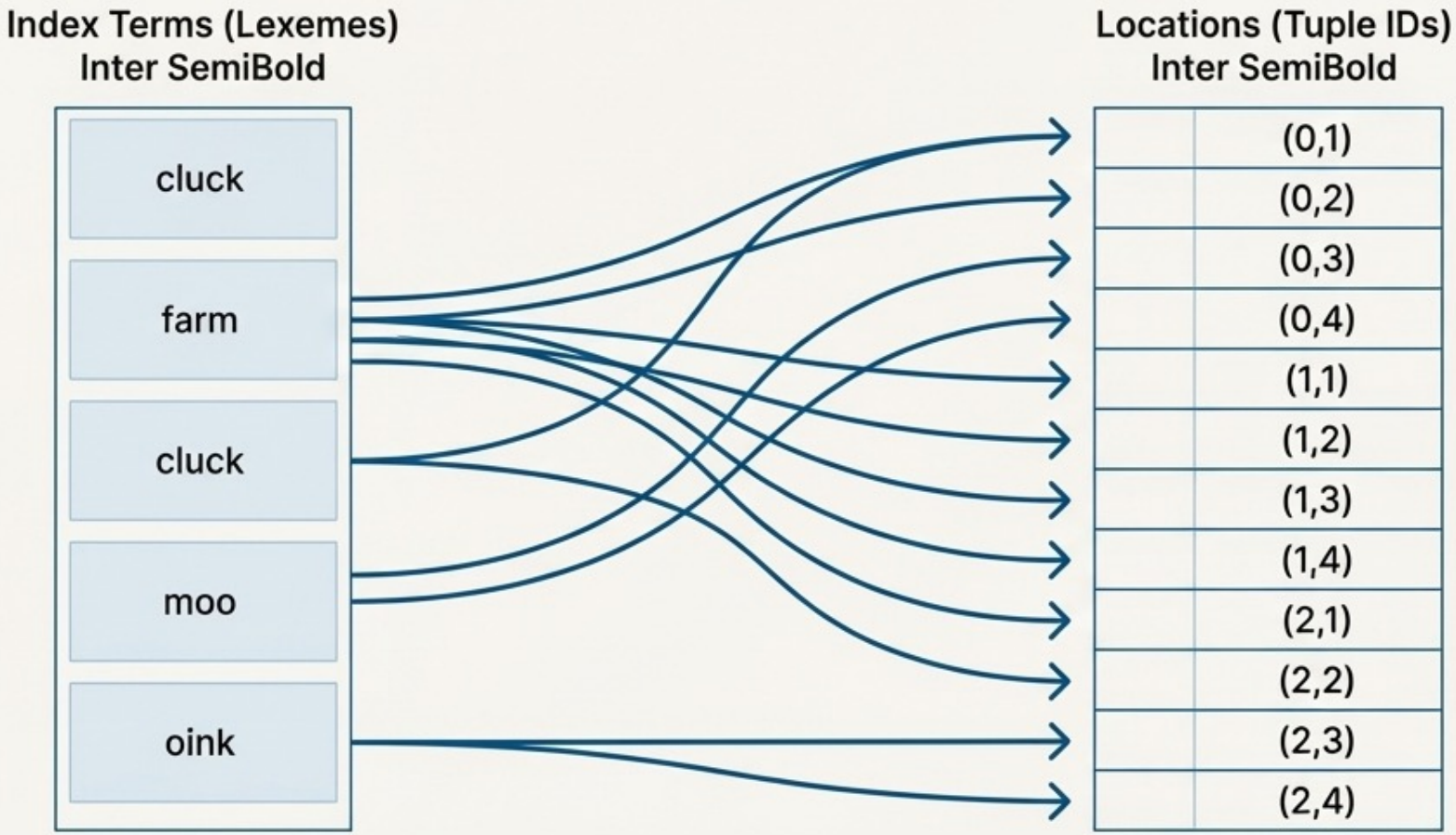
B-Tree indexes are highly efficient for atomic, sortable values (like integers or simple strings). However, their effectiveness diminishes when you need to search *inside* complex, non-atomic data types. A traditional B-Tree indexes the value as a whole, not its internal components.



NotebookLM

The GIN Philosophy: Index the Components, Not the Container

Instead of indexing the entire composite value, a GIN index deconstructs it into its constituent elements. It then creates a mapping from each individual element back to all the rows (identified by their Tuple ID or TID) that contain it. This is analogous to a book's index, which maps important terms to the page numbers where they appear.



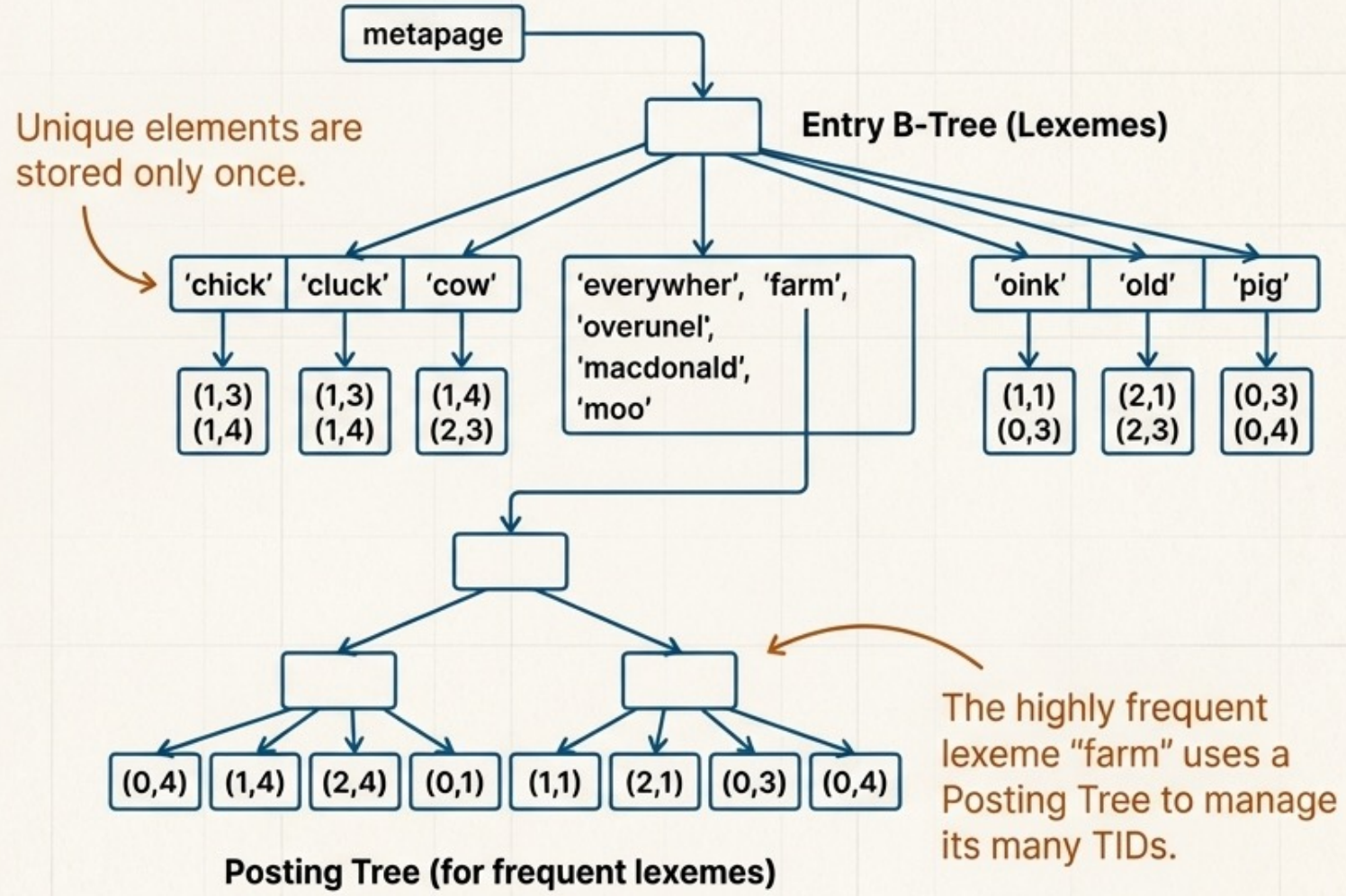
NotebookLM

The Anatomy of a GIN Index

A GIN index is a B-Tree where keys are the unique elements (e.g., lexemes). Each key is stored only once to save space.

Each leaf entry in this B-Tree points to a list of Tuple IDs (TIDs) called a “posting list.”

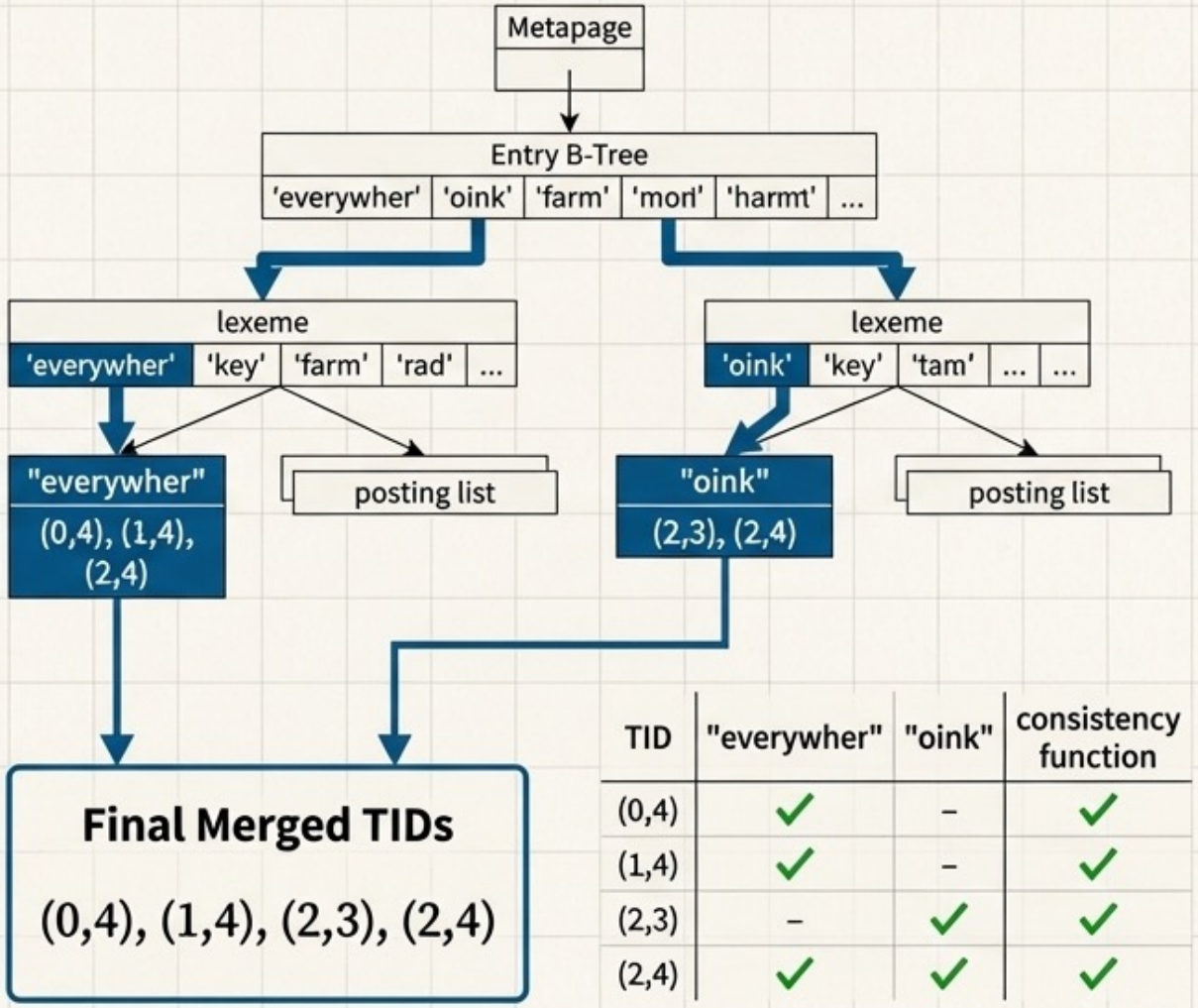
For very common elements with long posting lists, this list is stored in a separate, dedicated B-Tree called a “posting tree” for efficient traversal.



NotebookLM

Tracing a Query: `everywhere | oink`

1. **Extract Keys:** The engine's support functions extract the search keys ('everywher', 'oink') from the `tsquery'.
2. **Traverse Tree:** It traverses the Entry B-Tree to locate each key.
3. **Collect TIDs:** It retrieves the TIDs from the posting list of each found key. Because posting lists are sorted, multiple streams of TIDs can be efficiently merged into a single, sorted list.
4. **Check Consistency:** For each TID in the merged list, a 'consistency function' is called to verify if the row truly satisfies the full query logic (e.g., `OR` in this case).

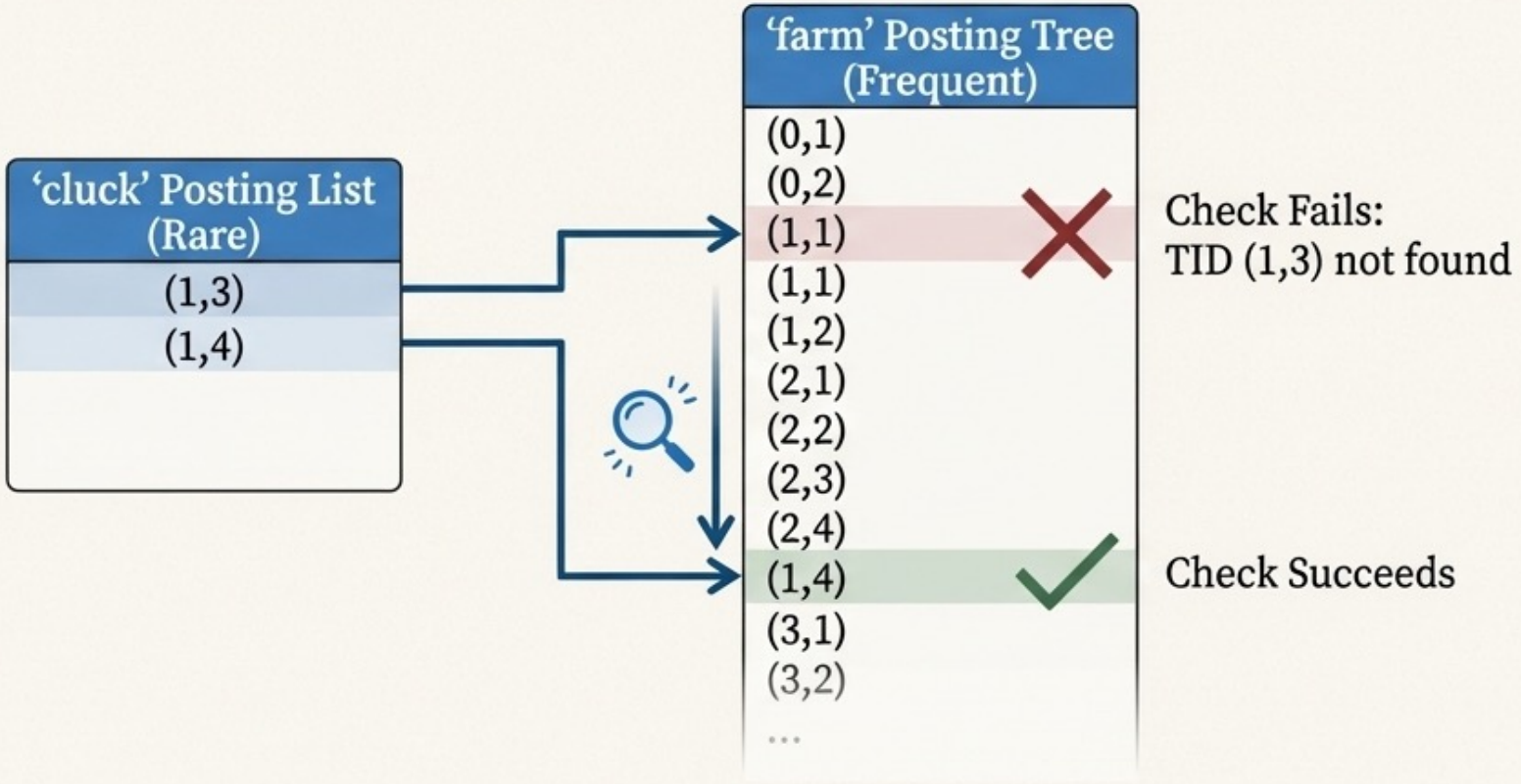


TID	"everywher"	"oink"	consistency function
(0,4)	✓	–	✓
(1,4)	✓	–	✓
(2,3)	–	✓	✓
(2,4)	✓	✓	✓

NotebookLM

The Smart Scan: Prioritize the Rare, Verify the Common

For `AND` queries with both rare and frequent terms (e.g., `farm & cluck`), GIN performs a significant optimization. Instead of scanning the massive posting tree for the common term ('farm'), it first finds the very few TIDs for the rare term ('cluck'). Then, it efficiently checks only those specific rows to see if they *also* contain 'farm.' This strategy dramatically reduces I/O and the number of consistency checks required.



Only two targeted lookups were performed on the large list; the vast majority was never scanned.

NotebookLM

The Update Challenge and the `fastupdate` Solution

Updating a GIN index can be slow. A single new document can touch many different keys in the index tree, leading to high I/O. To mitigate this, PostgreSQL uses a `fastupdate` mechanism (on by default). New index entries are first written to an unsorted, temporary "pending list." When this list reaches a size threshold (controlled by `gin_pending_list_limit`), its contents are efficiently sorted and merged into the main index in a single batch operation. This amortizes the cost of updates across many inserts.

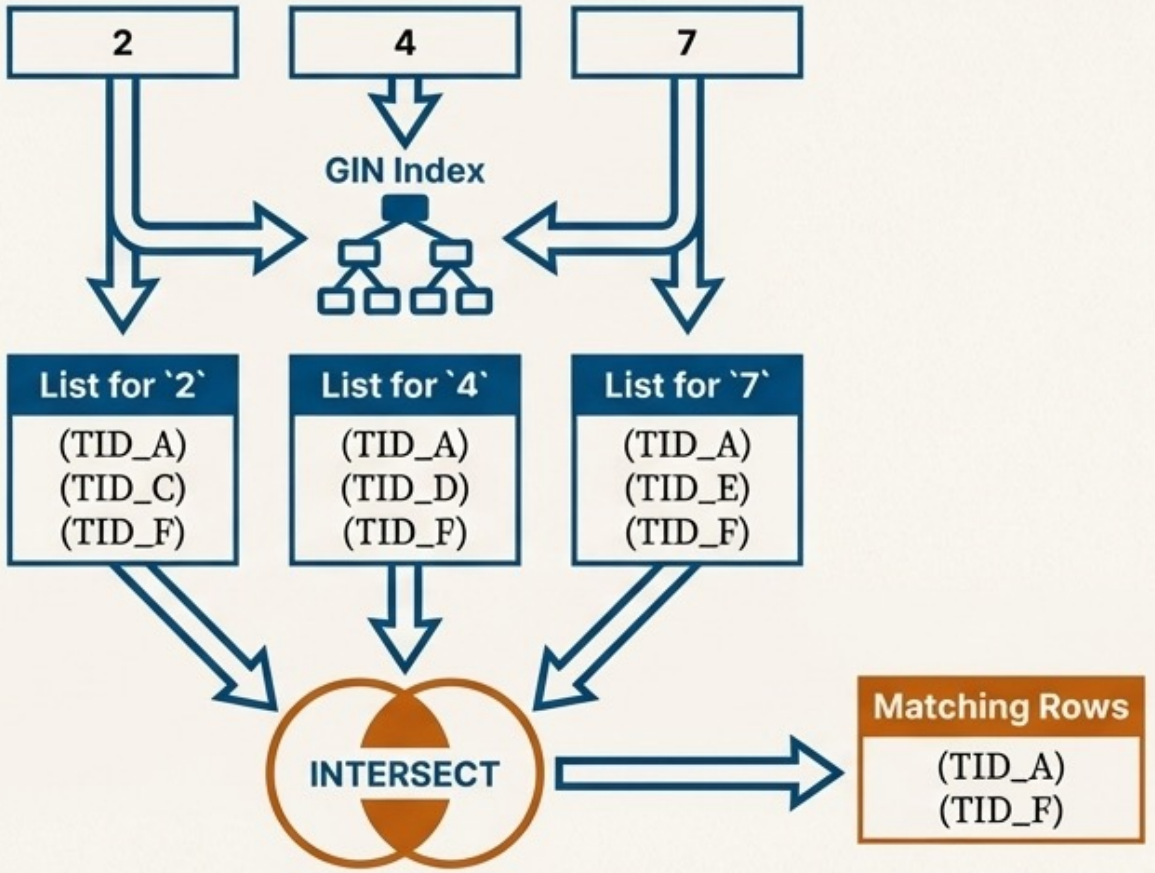


NotebookLM

Beyond Full-Text: Indexing Arrays for Containment and Overlap

The inverted index model is perfectly suited for array data types. A GIN index on an array column allows for extremely fast queries that check for element containment (@>`), being contained by (`<@`), or overlap (`&&`). The index stores each unique array element as a key, mapping it to the rows containing that element.

```
SELECT * FROM routes
WHERE days_of_week @> '{2, 4, 7}';
```



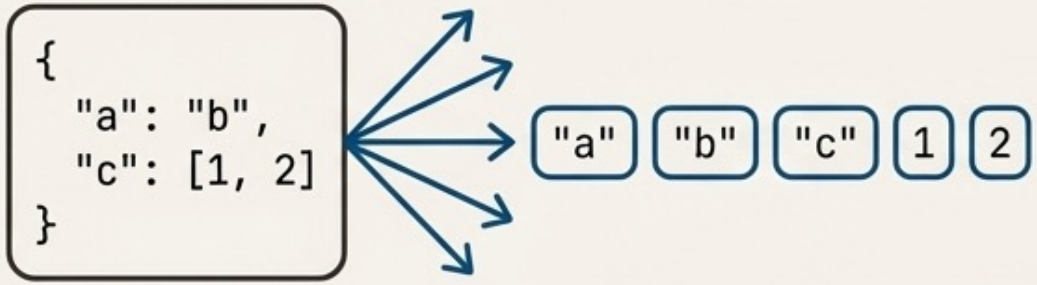
NotebookLM

Powering Modern Applications: Two Strategies for Indexing JSONB

GIN offers two distinct operator classes for `jsonb`, each creating a different kind of index with specific performance trade-offs. Choosing the right one is key to performance.

`jsonb_ops` (Default)

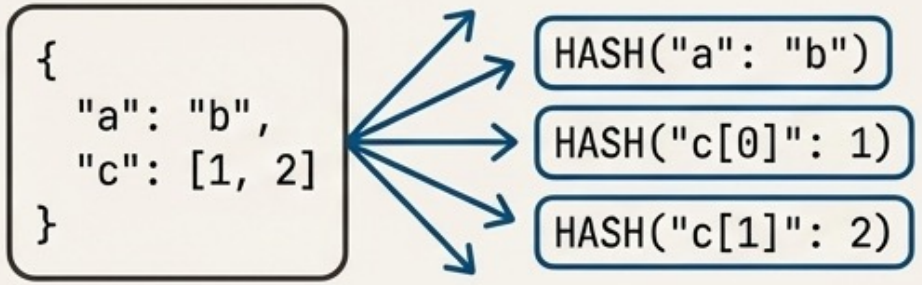
Indexes every key and primitive value as separate items.



Best for: Existence checks (`?`) and general containment (`@>`).

`jsonb_path_ops`

Indexes the hash of the full path to each value.

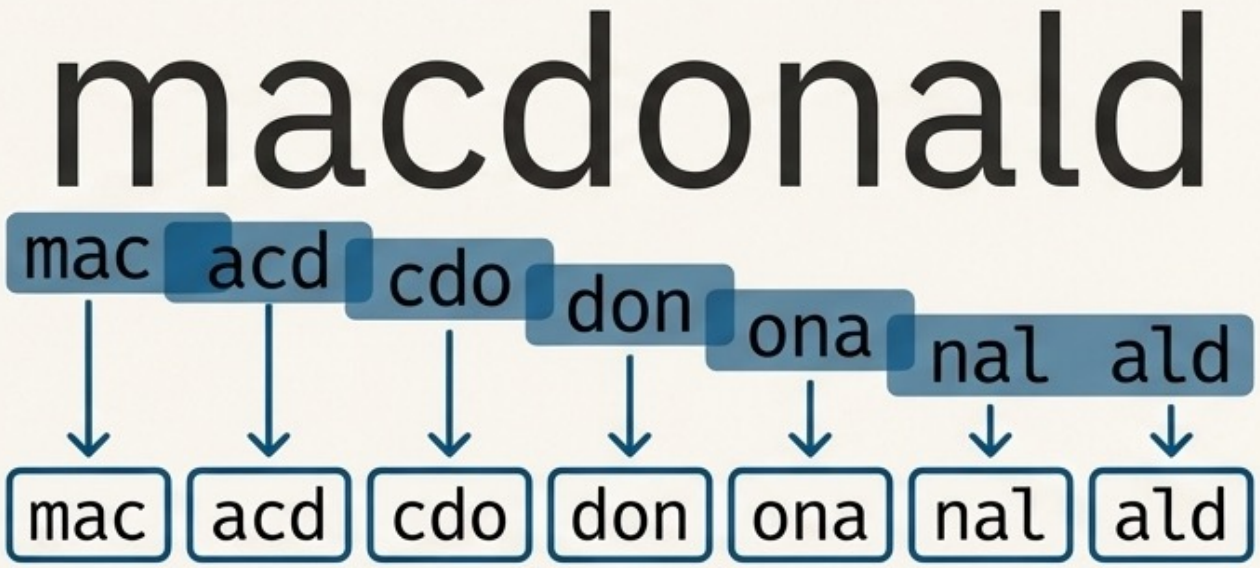


Best for: More precise, faster lookups for path-specific queries (`@@`, `@?`) as it avoids rechecks.

NotebookLM

Finding ‘MacDonald’ When They Type ‘McDonald’

The `pg_trgm` extension enables GIN to perform powerful similarity and fuzzy-string searches. It works by deconstructing text into a set of trigrams (3-character slices). A GIN index built with the `gin_trgm_ops` operator class indexes these trigrams. This allows PostgreSQL to very quickly find strings that share a high number of common trigrams, making `LIKE`, `ILIKE`, and similarity (`%`) searches extremely fast.



{ mac , acd , cdo , don , ona , nal , ald }

NotebookLM

A Specialist, Not a Generalist: Knowing GIN's Limitations

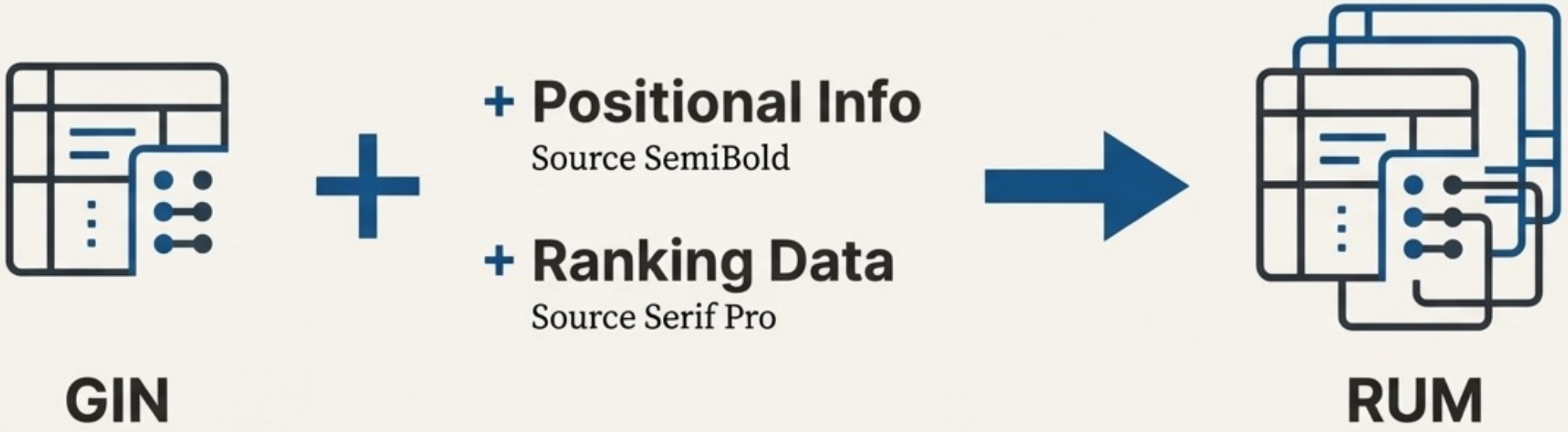
GIN is highly optimized for its specific task of indexing components within composite types. This specialization comes with important properties and limitations that influence its use.

✓ Supported Features	✗ Unsupported Features
Bitmap Scans: GIN always returns results as a bitmap. **Multi-Column Indexes**: Supported, but column order does not matter as it does in B-Trees.	**Index-Only Scans**: Not possible, as the original value is not stored in the index. **Ordering (`ORDER BY`)**: Cannot return results in a sorted order. **Uniqueness**: Cannot enforce uniqueness constraints. **Efficient `LIMIT`**: The entire result bitmap must be built before a `LIMIT` is applied, making it inefficient for finding the 'first N' rows.

NotebookLM

For Phrase Search and Ranking, Meet RUM

A key limitation of GIN is that it does not store positional information of lexemes within a document. This means it cannot natively accelerate phrase searches (e.g., “old macdonald”) or relevance ranking. For these advanced full-text search requirements, the **RUM** index extension, which is based on GIN, is the solution. RUM extends GIN by adding positional information to the index entries.



Trade-offs: Slower updates and larger index size.

NotebookLM

The GIN Index Decision Checklist

A practical guide for choosing the right index for your query patterns.

Use a GIN index for...

- ✅ High-performance full-text search (`tsvector @@ tsquery`).
- ✅ Checking for element existence, containment, or overlap in `array`, `hstore`, or `jsonb` (`@>`, `?`, `&&`).
- ✅ Accelerating fuzzy string matching (`LIKE`, `ILIKE`, `%`) with the `pg_trgm` extension.

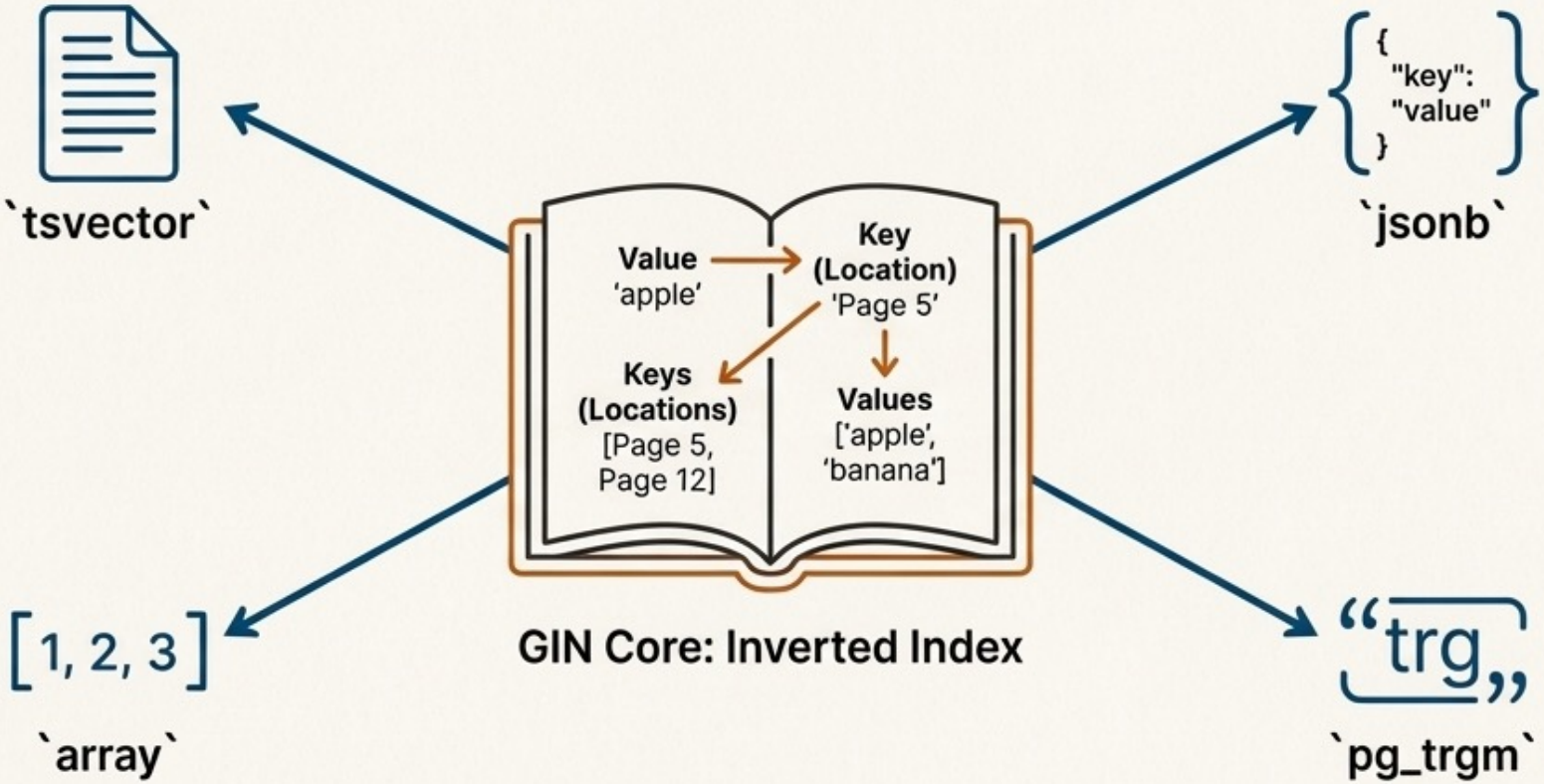
Consider Alternatives for...

- ❌ Simple equality or range queries on atomic types (Use **B-Tree**).
- ❌ Queries requiring sorted output (`ORDER BY`) (Use **B-Tree**).
- ❌ Advanced phrase search or relevance ranking in full-text search (Use **RUM**).

NotebookLM

From a Simple Idea on Idea to a Versatile Powerhouse

By inverting the relationship between a container and its components, GIN provides a powerful and extensible framework. It accelerates complex queries on the composite data types that are central to modern applications. Mastering GIN is essential for building high-performance, feature-rich systems with PostgreSQL.

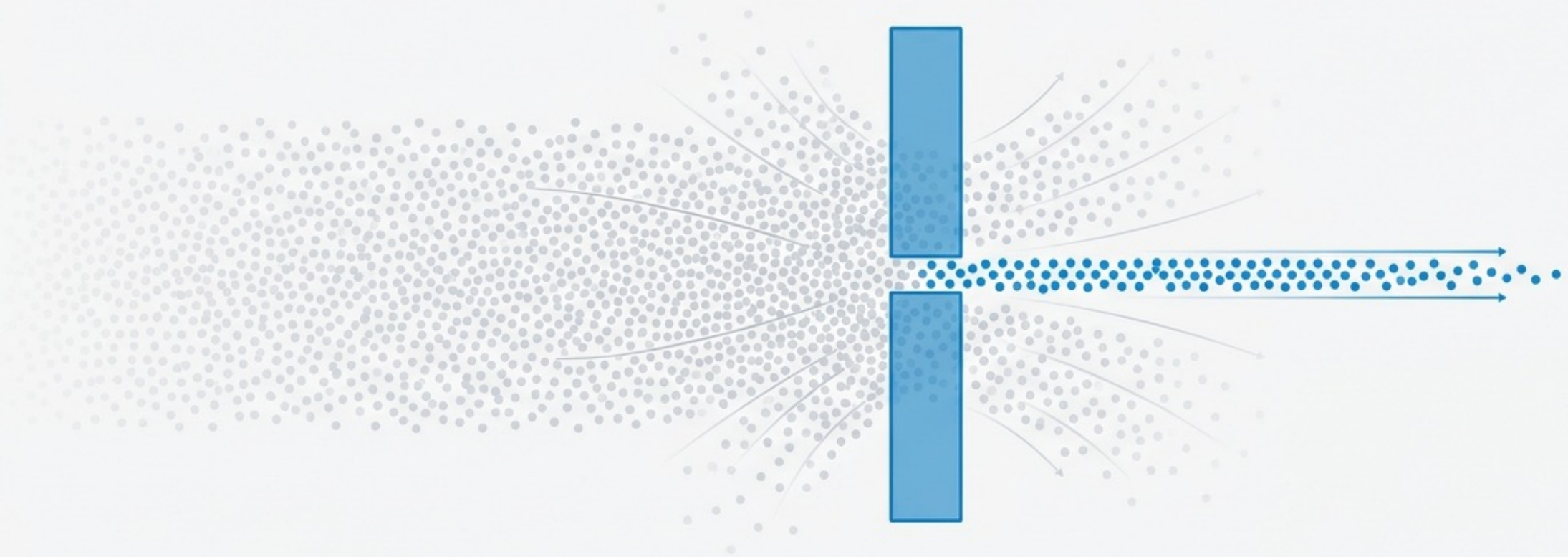


NotebookLM

29. BRIN

The Smart Filter: Conquering Terabyte Tables with PostgreSQL's BRIN Index

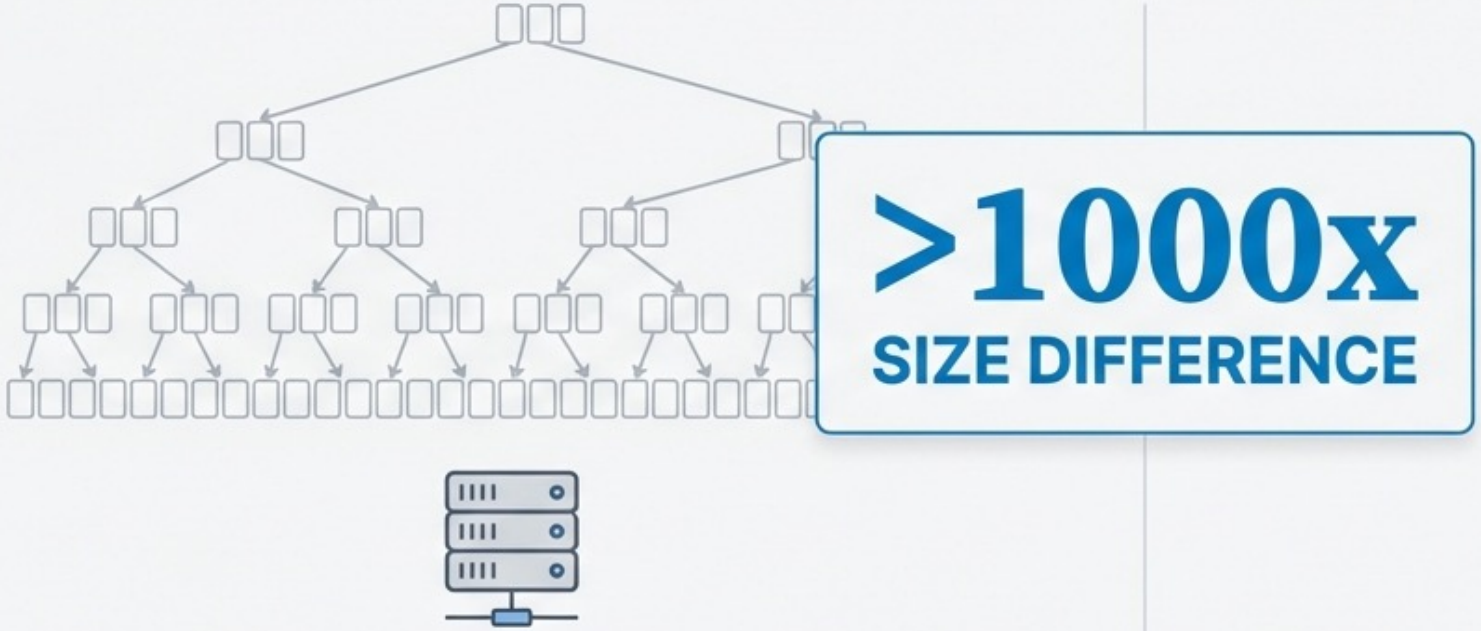
A deep dive into the philosophy, mechanics, and mastery of Block Range Indexes.



NotebookLM

The B-Tree Paradox: Indexing at Scale Creates a Scaling Problem

For Very Large Databases (VLDBs), the trusted B-Tree index can become a liability. As tables grow into terabytes, the index itself can consume hundreds of gigabytes, straining storage and memory.



>1000x
SIZE DIFFERENCE





B-Tree Index (`scheduled_time`): **210 MB**

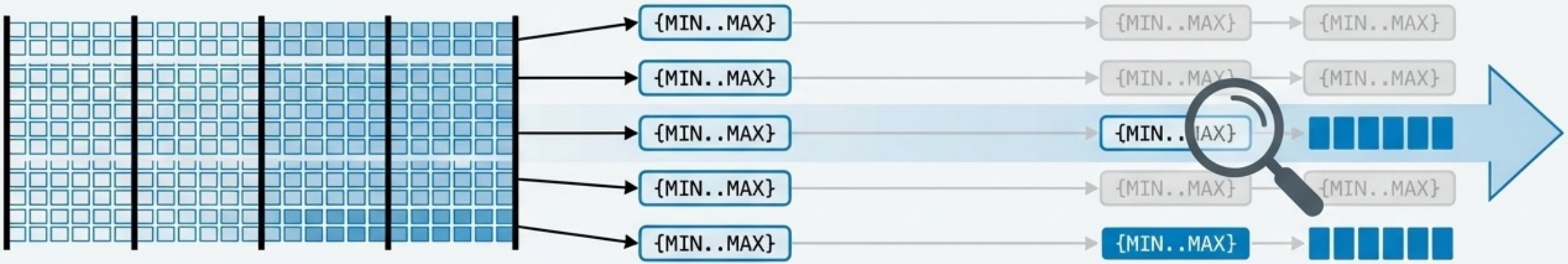
BRIN Index (`scheduled_time`): **184 kB**

That's a **>1000x** size difference. For massive tables, this isn't a minor optimization; it's a fundamental architectural choice.

NotebookLM

A New Philosophy: Summarize, Don't Point

Unlike a B-Tree which meticulously maps individual values to row locations (TIDs), a Block Range Index (BRIN) takes a different approach. It is optimized to **filter out irrelevant data** rather than pinpointing exact rows.



1. **Divide**: The table is logically divided into "block ranges" (groups of pages, default is 128 pages per range).

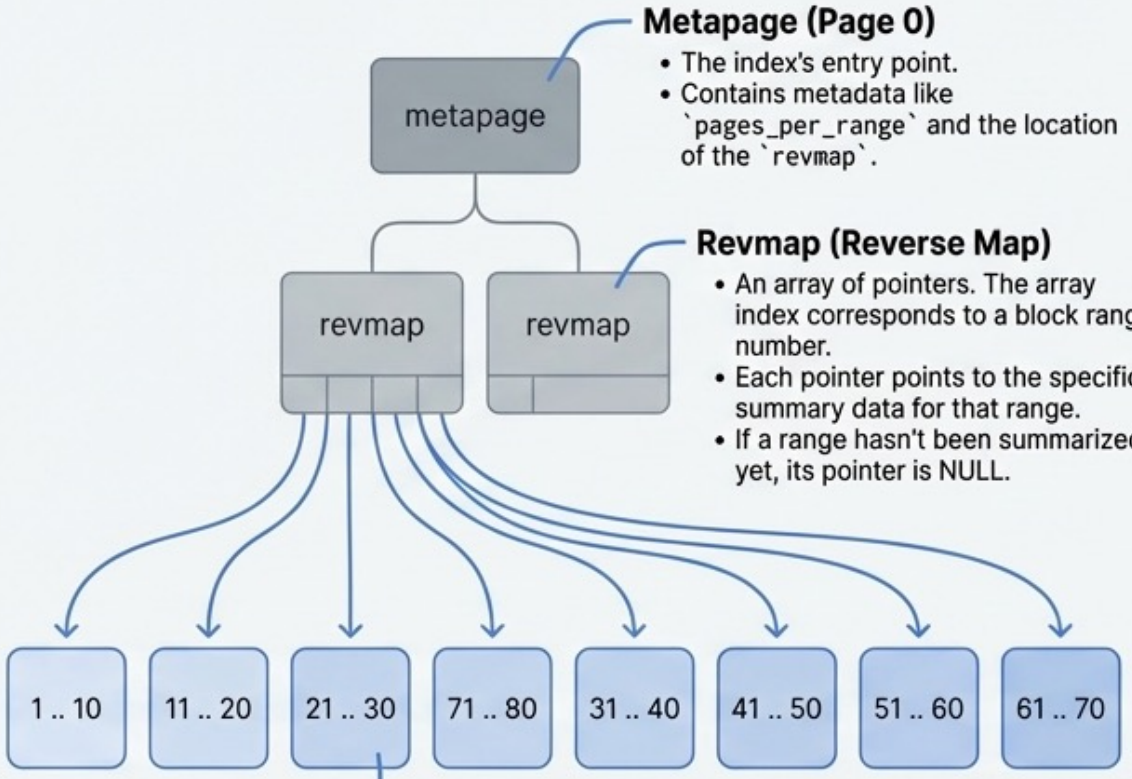
2. **Summarize**: For each range, BRIN stores a small summary of the data within it (e.g., the minimum and maximum values).

3. **Filter**: During a query, the planner checks the summaries. If a range's summary proves it cannot contain matching data, the entire range is skipped.

Key takeaway: BRIN acts as a 'smart filter' or an accelerator for sequential scans, not a traditional index in the B-Tree sense.

NotebookLM

The Anatomy of a BRIN Index



Metapage (Page 0)

- The index's entry point.
- Contains metadata like `pages_per_range` and the location of the `revmap`.

Revmap (Reverse Map)

- An array of pointers. The array index corresponds to a block range number.
- Each pointer points to the specific summary data for that range.
- If a range hasn't been summarized yet, its pointer is NULL.

Summary Data Pages

- Contain the actual index entries.
- Each entry is a summary for a specific block range. For an ordered type, this would be `{min_value .. max_value}`.

Under the Hood

We can inspect the structure directly:

```
-- Get metadata from page 0
SELECT pagesperrange, lastrevmappage
FROM brin_metapage_info(get_raw_page('flights_bi_scheduled_time_idx', 0));
-- Result: 128 | 4
```

```
-- See summary for the first three ranges
SELECT blknum, value
FROM brin_page_items(get_raw_page('flights_bi_scheduled_time_idx', 6), '...')
LIMIT 3;
-- Result:
-- 0    | {2016-08-15 02:45:00+03 .. 2016-08-15 16:20:00+03}
-- 128  | {2016-08-15 05:50:00+03 .. 2016-08-15 18:55:00+03}
-- 256  | {2016-08-15 07:15:00+03 .. 2016-08-15 18:50:00+03}
```

NotebookLM

The Search: A Lossy Bitmap Scan in Action



1. Query: A query with a WHERE clause on an indexed column arrives.

```
...
WHERE scheduled_time > '2016-08-15 17:00:00'
```

2. Scan Summaries

{min: 02:45, max: 16:20}	SKIP
{min: 05:50, max: 18:55}	INCLUDE
{min: 10:00, max: 18:55}	SKIP
...	
{min: 15:55, max: 20:20}	SKIP

3. Consistency Check



4. Build Bitmap

The page numbers for all *consistent* ranges are added to a bitmap. This bitmap is "lossy" because it flags entire pages, not individual tuples.



5. Heap Scan

The bitmap is used to scan the table, but only the pages flagged in the bitmap are read. Rows on these pages are re-checked against the query condition.

The heap pages are read **sequentially** block range by block range, leveraging pre-fetching for high throughput.

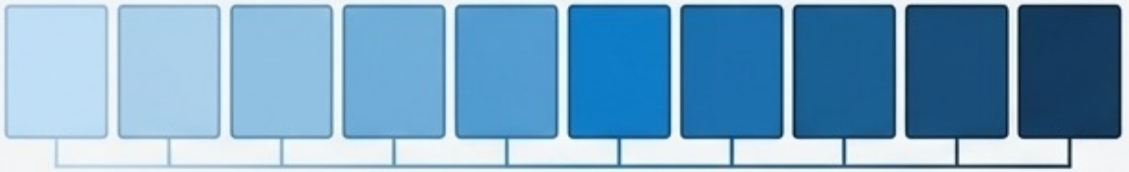
NotebookLM

The BRIN Toolbox: Mastering the `minmax` Operator Class

The effectiveness of a BRIN index depends entirely on choosing the right “operator class” for your data’s physical layout. The default and most common is `minmax`.

The Key to Success: Correlation

The `minmax` class works best when there is a high correlation between the physical location of rows and their logical values. In simple terms: values should be well-ordered on disk.



Why it works for `flights_bi`: New data is appended in chronological order and is rarely updated or deleted. This naturally creates high correlation.

Finding a Candidate

We can check this in the `pg_stats` catalog.

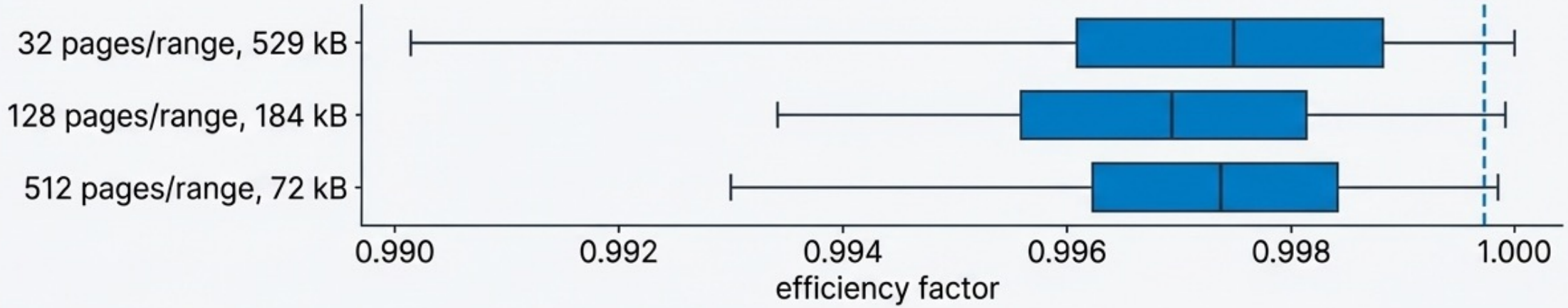
```
SELECT attname, correlation
FROM pg_stats
WHERE tablename = 'flights_bi'
ORDER BY correlation DESC;
```

```
     attname      |  correlation
------------------+---------------
 scheduled_time   |    0.99999949   <-- Perfect Candidate
 actual_time      |    0.99999948
 fare_conditions  |    0.79766897
 ...              |
 flight_no        |  0.0020146023   <-- Poor Candidate
 passenger_id     | -0.0004612129
```

NotebookLM

`minmax` in Action: High Efficiency, Low Overhead

When querying for a single day's worth of flights on the highly correlated `scheduled_time` column, the BRIN index is exceptionally efficient.



The Query Plan

A `Bitmap Heap Scan` reads only the pages identified by the index.

- Pages Read: `Buffers: shared hit=1561`
- Total Table Pages: 528,417

Efficiency Factor

99.7%

(528,417 - 1561) / 528,417

We filter a 4GB table and find all relevant rows by reading just **0.3%** of the table's pages. The chart shows this high efficiency is maintained even with much larger range sizes (and thus smaller indexes).

NotebookLM

The Outlier Problem: When Data Correlation Breaks Down

The Scenario

- What happens if we disrupt the natural order?
  1. `DELETE` a small fraction of random rows (e.g., 0.1%).
  2. `VACUUM` the table, creating free space throughout the file.
  3. `INSERT` new, chronologically later data. These new rows will fill the vacuumed gaps instead of being appended to the end.

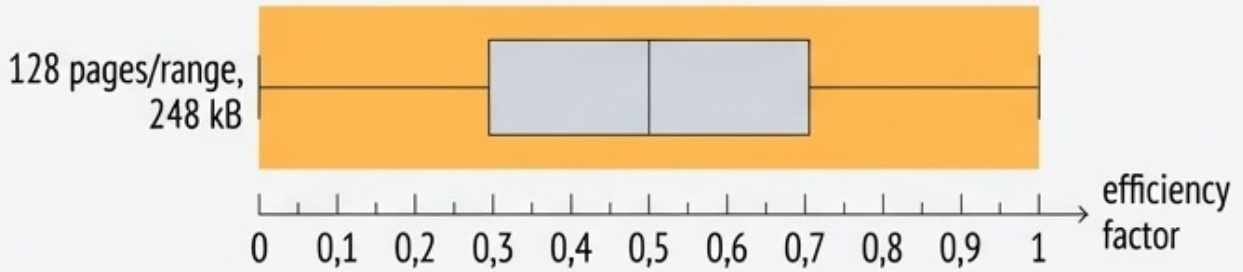
The Impact

BEFORE: {2016-08-15 ... 2016-08-15}

AFTER: {2016-08-15 ... 2017-08-16} — Now uselessly wide

The Devastating Result

Index efficiency collapses. The index now flags almost every range as potentially relevant, forcing a near-full table scan.



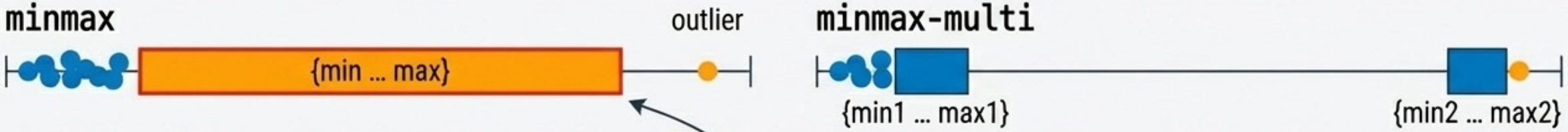
NotebookLM

The Solution for Chaos: `minmax-multi`

The Big Idea

Instead of storing one single **{min .. max}** summary that is vulnerable to outliers, `minmax-multi` stores a *list* of smaller summary ranges.

How it Works (visual explanation)



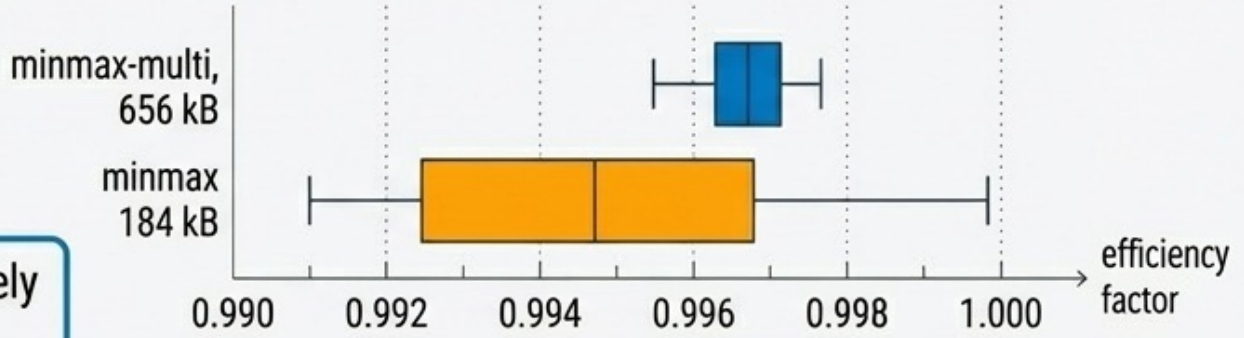
Vulnerable to outliers: A single extreme value stretches the summary, making it uselessly wide.

Robust to outliers: A new value simply creates a new, contained summary range, leaving others untouched.

The `values_per_range` parameter controls the maximum number of summary intervals stored per block range.

The Comeback (main visual)

By replacing the `minmax` index with `minmax-multi`, we restore the high efficiency, even on the 'dirty' data.



Key Takeaway: Efficiency is restored at the cost of a moderately larger index (656 kB vs 184 kB), a worthwhile trade-off.

NotebookLM

Specialized Tools: `inclusion` for Spatial and Range Data

Use Case

For data types that don't have simple linear ordering but do have containment properties (like geometric shapes or IP network ranges). The `inclusion` operator class stores a "bounding box" summary of all values in the range.

Indexing the `airport_coord` (a `point` type) in our `flights_bi` table. We can use this to find all flights within a geographic box.



A Critical Caveat: The Planner's Blind Spot ⚠️

PostgreSQL does not collect correlation statistics for most non-scalar types.

Without these statistics, the query planner assumes the worst and estimates the index scan cost to be very high.

Result: The planner will almost always prefer a sequential scan, even when the BRIN index would be much faster. You may need to force its hand with SET `enable_seqscan = off;`.

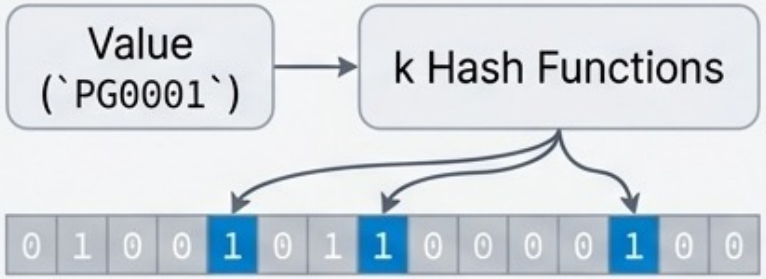
Key Takeaway: `inclusion` can be very powerful, but be aware that you might have to fight the planner to use it.

NotebookLM

Specialized Tools: `bloom` for Low-Correlation Data

What about columns with no natural ordering, like `flight_no`? The `bloom` **operator class** uses a Bloom filter to create a summary.

Bloom Filter Refresher



- A compact, probabilistic data structure to test set membership.
- **Allows false positives**: It might say a value is in the set when it's not.
- **No false negatives**: It will never say a value is *not* in the set when it is.
- For BRIN, the 'set' is the collection of unique values within a block range.

Tuning Parameters

- **`n_distinct_per_range`**: An estimate of the number of unique values in a range. A negative value is treated as a percentage of total rows.
- **`false_positive_rate`**: Controls the size and accuracy of the filter. Lower values create larger, more accurate filters.

Supported Operations

- Only equality (`=`) is supported, as Bloom filters are based on hashing.

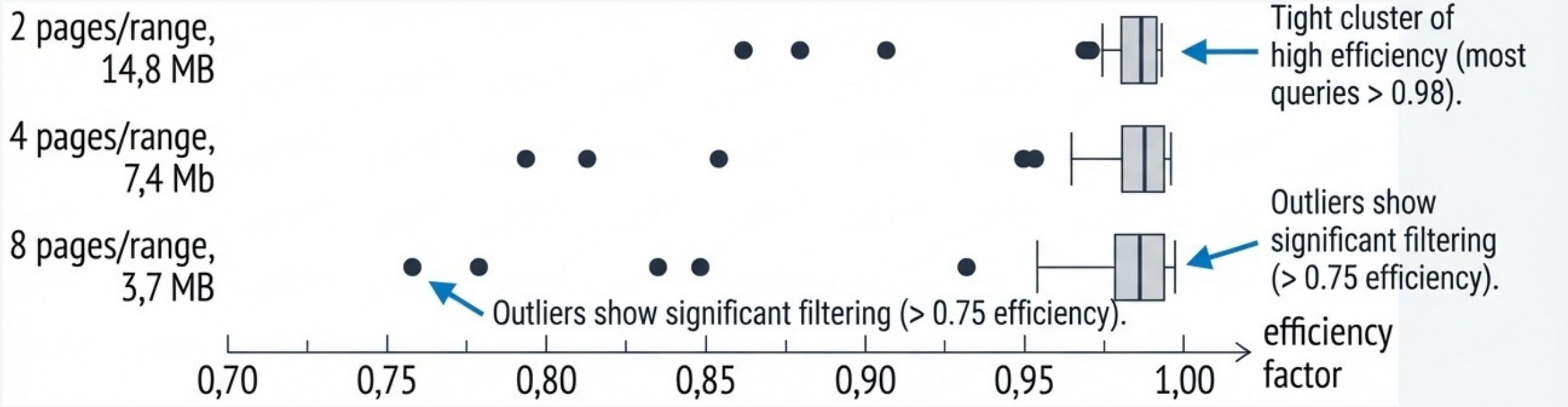
Key Takeaway: The 'bloom' operator class is ideal for columns with many distinct values and low correlation, providing a compact summary with tunable accuracy.

NotebookLM

“bloom” in Action: Finding Needles in a Haystack

The Challenge: The `flight_no` column has a correlation near zero (0.002), making it useless for `minmax`. Can BRIN still help?

The Result: By creating a `bloom` index, we can still achieve very high efficiency when searching for a specific flight number.



Conclusion: For low-correlation, high-cardinality equality checks on very large tables, a BRIN `blooom` index can be an unexpectedly powerful tool.

NotebookLM

The BRIN Index: Properties and Limitations

Access Method Properties

`pg_indexam_has_property`

- `can_unique: f` → **Cannot** be used for `UNIQUE` or `PRIMARY KEY` constraints.
- `can_order: f` → **Cannot** provide sorted output (no index-only scans that avoid a sort).
- `can_multi_col: t` → **Supports** multi-column indexes. Useful if all columns share the same optimal `pages_per_range`.
- `can_exclude: f` → **Cannot** be used for `EXCLUDE` constraints.

Index-Level Properties

`pg_index_has_property`

- `bitmap_scan: t` → The **only supported access method** is a bitmap scan.
- `clusterable: f` → `CLUSTER` is not supported. While reordering data helps BRIN, the operation is considered too costly for the target VLDBs. Natural data ingestion order is preferred.
- `search_nulls: t` → `IS NULL` and `IS NOT NULL` queries **are** supported.

NotebookLM

The BRIN Playbook: When to Use It

Create a BRIN index when your scenario matches this profile:



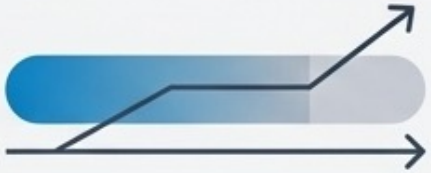
Table Size: Massive

Hundreds of gigabytes, terabytes, or more. The benefits are most pronounced at extreme scale.



Workload: Append-mostly

Data is primarily inserted, not updated or deleted. Think data warehouses, time-series data, logs, and event streams.



Data Correlation: High

There is a strong, natural correlation between the physical storage order of rows and the values in the indexed column(s).

Examples: Timestamps, auto-incrementing IDs.



Query Patterns: Filtering large chunks

Queries that select wide slices of data (e.g., 'all events from last week') benefit most from skipping large ranges.

Or... if your data has low correlation but you can leverage a specialized operator class like bloom for equality checks.

NotebookLM

A Specialized Tool for a Specialized Problem

BRIN is not a replacement for the B-Tree. The B-Tree remains the default, general-purpose index for most OLTP workloads.



General Purpose & Specialized Tool

BRIN is a specialized, hyper-efficient tool designed to conquer the unique indexing challenges of very large databases. By trading pinpoint accuracy for massive space savings and high-speed filtering , it offers a powerful solution for the right workload.

When you're managing terabytes, think differently. Think in ranges. Think BRIN.

NotebookLM

Thank You

Questions?

NotebookLM