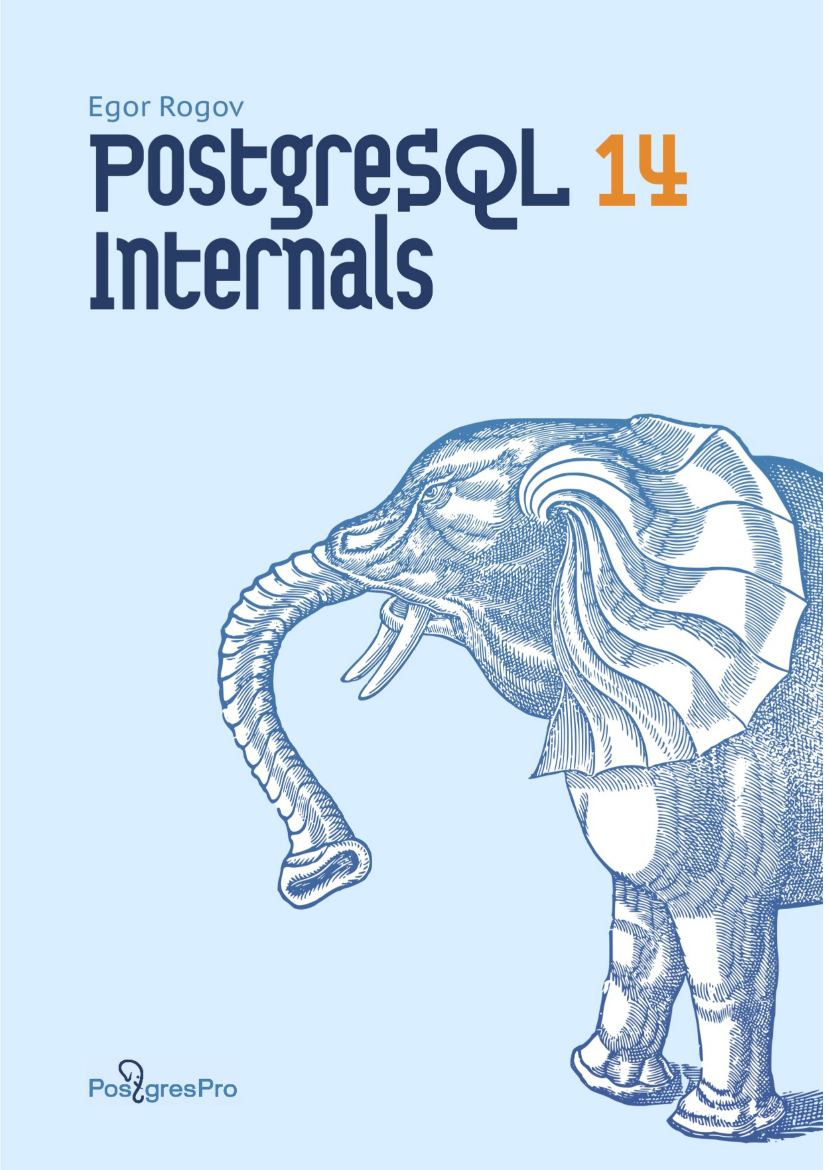
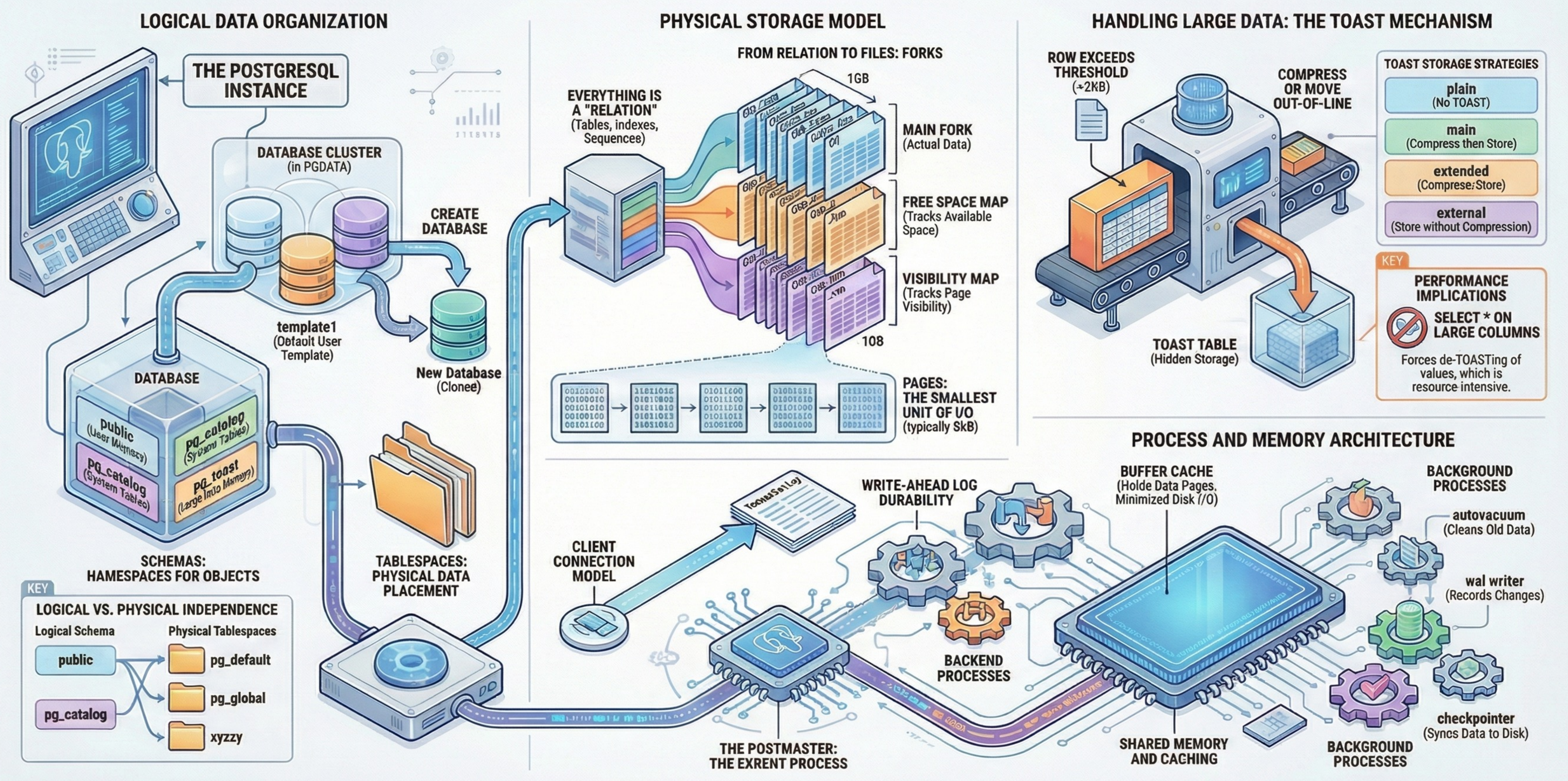
InfoGraph



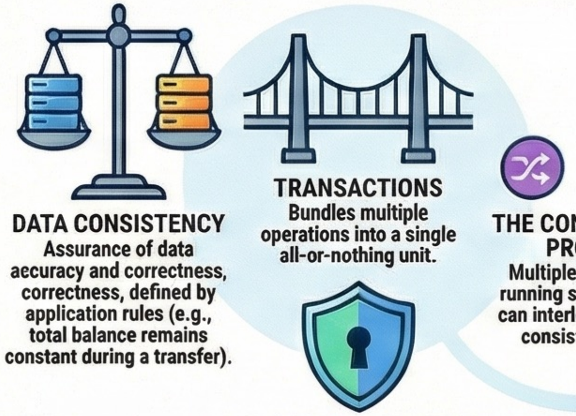
Under the Hood: The Architecture of PostgreSQL



NotebookLM

A Visual Guide to Database Transaction Isolation

1. The Core Challenge: Consistency vs. Concurrency



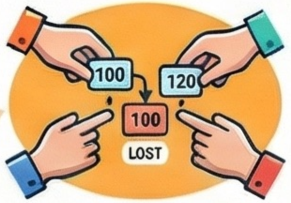
DATA CONSISTENCY
Assurance of data accuracy and correctness, correctness, defined by application rules (e.g., total balance remains constant during a transfer).

TRANSACTIONS
Bundles multiple operations into a single all-or-nothing unit.

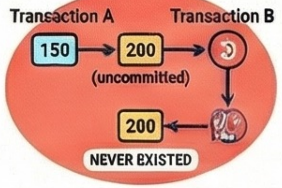
THE CONCURRENCY PROBLEM
Multiple transactions running simultaneously can interlence, sIolating consistency rules.

THE NEED FOR ISOLATION
To guarantee consistency, transactions must be Isolated, ensuring concurrent execution equals sequential execution.

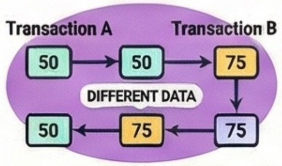
2. Concurrency Anomalies: What Can Go Wrong?



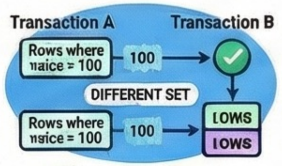
Lost Update
Two transactions read and update the same value, everwriting and "losing" the first's change.



Dirty Read
Transaction A reads uncommitted data from Transaction B. If B rulle back, A read Invalid data.



Non-Repeatable Read
A transaction reads a row twice but gets different data each time because another transaction modified and committed that row in between the two reads.



Phantom Read
A transaction runs the same query twice but gets a different set of rows because another transaction inserted or deleted rows that match the query's criteria.

3. The S· The SQL Standard: Four Levels of Isolation

Read Uncommitted
Most lenient. Choosing a level is a trade-off between consistency guarantees and performance.

Read Committed

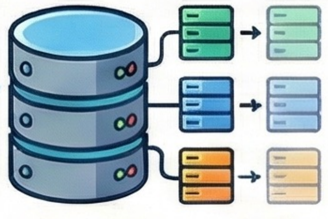
Repeatable Read

Serializable
Stricter levels often require more looking, reducing concurrency.

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

Note: "—" indicates the anomaly is prevented. "Yes" indicates it is allowed.

4. The PostgreSQL Way: Snapshot Isolation



PostgreSQL Uses Snapshot Isolation
Employe Multi-Version Concurrency Control (MVCC). Each transaction operates on a consistent "snapshot" from its start, reducing locking.

Stricter Than The Standard
In PostgreSQL, Dirty Reads are never permitted. Read Uncommitted behaves exactly like Read Committed.

Beware of New Anomalies
Prevents standard anomalies but can suffer from Read Shew (in Read Committed) and Write Show (in Repeatable Read).

Serialization Failures Instead of Anomalies
At higher levels (Repeatable Read, Serializable), PostgreSQL proactively aborts conflicting transactions.

PostgreSQL Level	Lost Updates	Dirty Reads	Non-Repeatable Reads	Phantom Reads	Other Anomalies
Read Committed	✓	✗	✓	✓	Yes
Repeatable Read	✗	✗	✓	✗	Yes
Serializable	✗	✗	✗	✗	—

Note: "Lost Updates" refers to application logic reading, calculating, and writing back, not prevented by default.

5. Practical Guidance: Which Level Should You Use?

Read Committed (The Default)

PROS:
Good performance, no forced transaction retries.

CONS:
Developer is responsible for preventing anomalies like read skew and lost updates using explicit locks (SELECT FOR UPDATE) or single atomic SQL statements.

Repeatable Read

PROS:
Protects against more anomalies, great for complex, multi-statement read-only reports.

CONS:
Application must have logic to retry transactions that fail due to serialization errors. Still vulnerable to write skew.

Serializable (The Safest)

PROS:
Guarantees full data consistency, simplifying application logic.

CONS:
Highest performance overhead, requires rebust transaction retry logic, and cannot be used with read replicas.

Inside a PostgreSQL Page: A Visual Guide to Data Storage and Transactions

Anatomy of a PostgreSQL Page (8KB Block)

Every page follows a specific five-part structure.

Data is stored in pages, which are divided into a header, pointers, the actual data (items), free space, and a special area.

Header (PageHeaderData)
Fixed 24 bytes. Metadata, checksums, pointers (lower, upper, special)

Item Pointers (ItemIdData)
Array of 4-byte pointers to tuple locations. Enables tuple movement without breaking references

Free Space
Empty contiguous block where new tuples and pointers grow towards each other

Items (Tuples)
Actual row data (row versions). Added from end of page backwards

Special Space
Located at end. Used for index-specific information (e.g., 8-tree), often empty

The Structure of a Row Version (Tuple)

23+ Byte Header (MVCC Metadata)

- **xmin:** Transaction ID that inserted
- **xmax:** Transaction ID that deleted/updated (0 if current)
- **ctid:** Pointer to next version
- **infomask:** Status/hint bits (e.g., committed/ aborted)

User Data

boolean | integer | ... | ...

Data Alignment Wastes Space

Order affects size due to padding

Inefficient (40 bytes)
boolean | integer | boolean | integer
6 bytes wasted

Efficient (34 bytes)
integer | integer | boolean | boolean
Optimized storage

The Lifecycle of a Tuple: MVCC in Action

1. INSERT

xmin: 778 (Current)
xmax: 0

New tuple added. xmax set to 0, indicating current, valld version

2. COMMIT

xmax: 0

Commit Log (CLOG)
Tan 778: Committed
...

Status recorded in external Commit Log. Tuple on page is not immediately changed

Hint Bits Are Set Lazily

amin_committed_set ✓

Next reader checks CLOG, then sets `xmin_committed` hint bit to avoid future lookups

3. UPDATE

xmax: 778 (Current)

xmin: 778 (Current)
xmax: 0
ctid: man: 778 (Current)

Two-stop process: Old tuple `xmax` set to current transaction ID. New tuple created with `xmin` as current ID

4. DELETE

xmax: 778 (Ooleting)
xmax: 0

Tuple is not physically removed. `xmax` set to ID of dolating transaction. Becomes invisible after commit

5. ROLLBACK (Abort)

xmax: 778 (Aborted)

Commit Log (CLOG)
Tan 778: Aborted
...

Transaction marked aberted in CLOG. Changes remain on page; `xmax_abcrted` hint bit will be set by next observer, making change invisible

Supporting Structures & Concepts

Indexes Point to All Row Versions

Index → Page 1 (TIDs, TIDs, TIDs) → Page 2 (..., ..., ...)

Indexes lack `amin/weax`. index entries point to specific TIOs, but table page must be checked for visibility

TOAST for Large Data

Main tuple tuple → TOAST Pointer → TOAST Table

Large column data moved to separate TOAST table. Main tuple holds a pointer

Virtual Transactions Optimize Reads

Read-Only Transaction → Virtual ID
Modifying Transaction → Permanent ID

Read-only transactions get virtual IDs; permanent IDs assigned only if modifying data

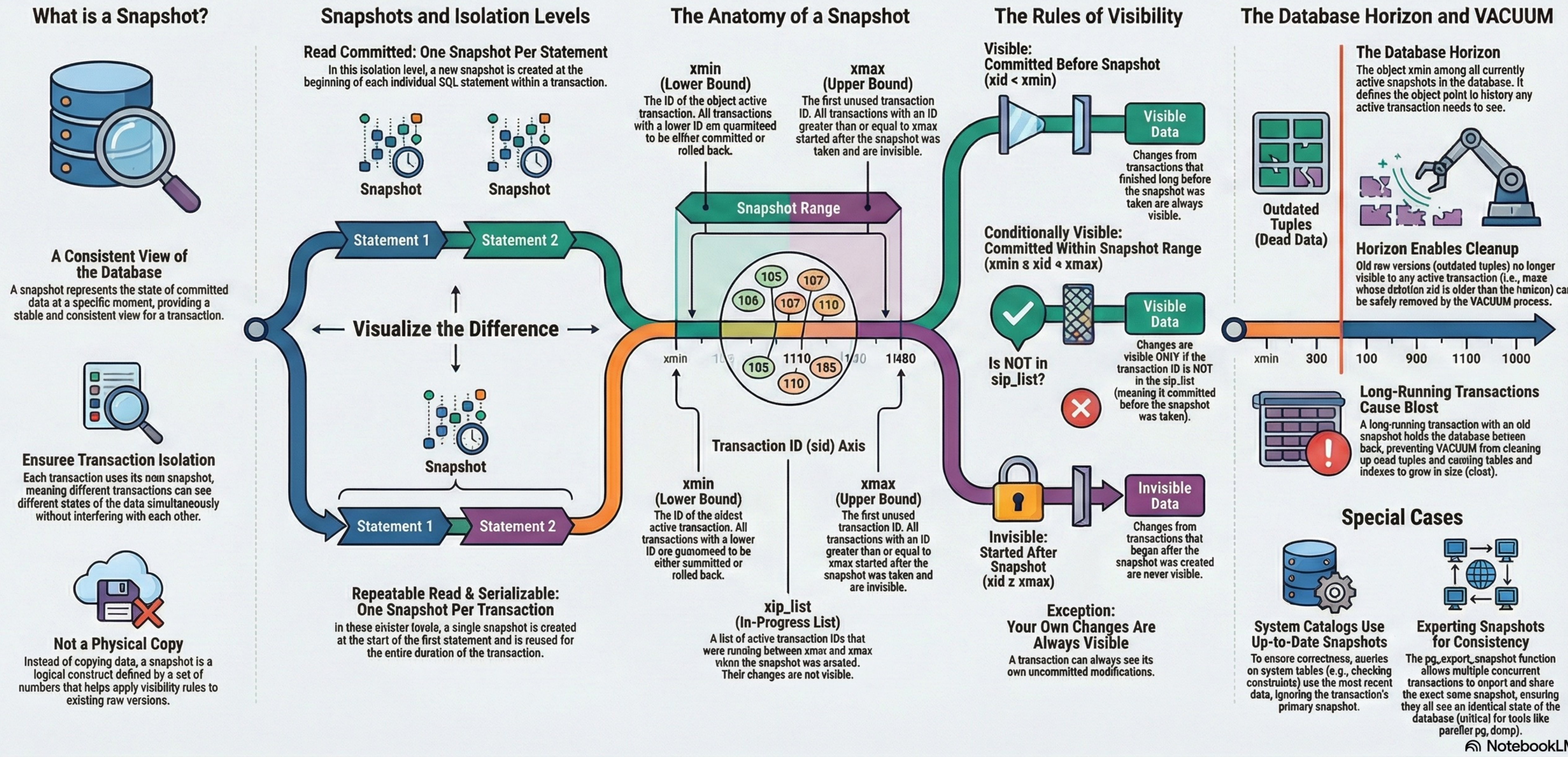
Subtransactions for Savepoints

Parent → Txn A, Txn B, Txn C

Using `SAVEPRINT` creates subtransactions with own IDs. Commit status depends on parent, tracked in `pg_subtrans`

NotebookLM

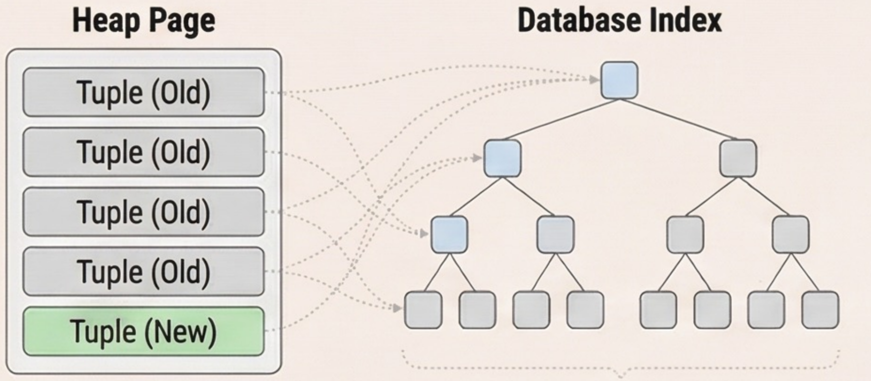
A Deep Dive into PostgreSQL Snapshots: Understanding Transaction Visibility



Anatomy of a PostgreSQL Update: How HOT & Page Pruning Optimize Performance

The Problem: Standard Updates Are Expensive

Every UPDATE creates a new tuple version, leaving the old one behind. Every index accuulze pointers to "dead" bot tuples over time.

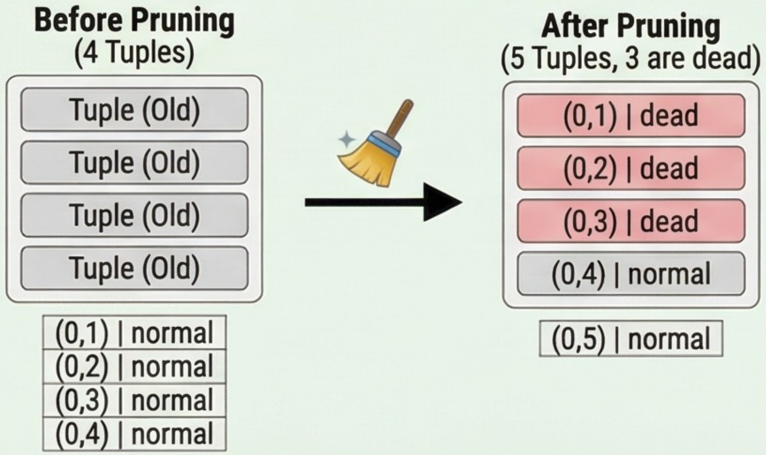


Every index must be updated, even for changes to non-indexed columns. Indexes accumulate pointers to "dead" tuples over time.

First Line of Defense: Page Pruning

What is Page Pruning?

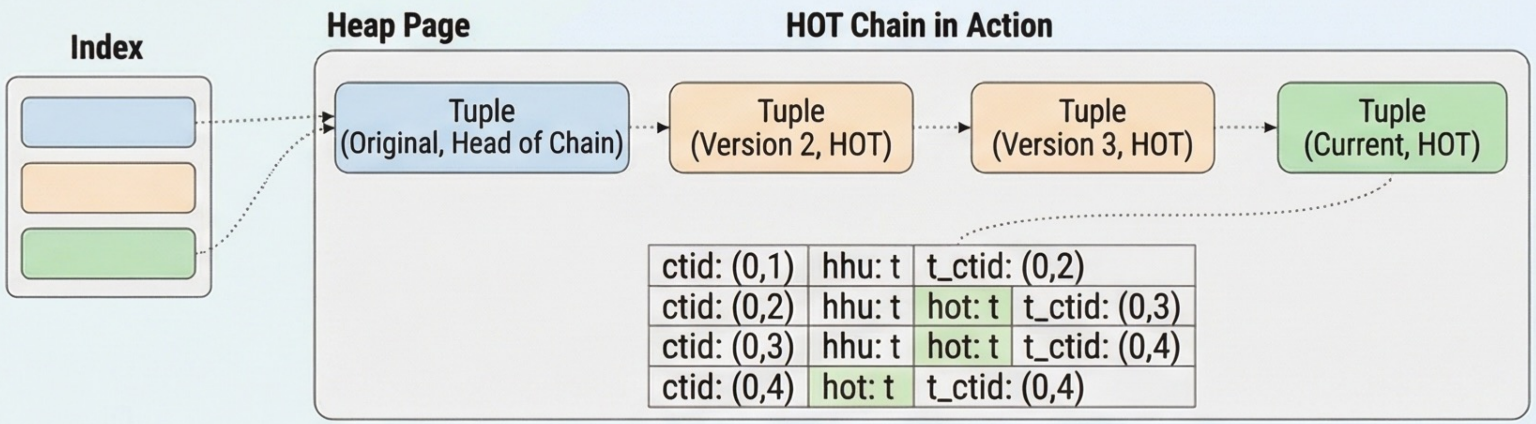
A fast, automatic cleanup that removes dead tuples (no longer visible to any transaction) from a single heap page.



The Ultimate Optimization: Heap-Only Tuple (HOT) Updates

What are HOT Updates?

An optimization that avoids creating new index entries when an UPDATE does not modify any indexed columns.



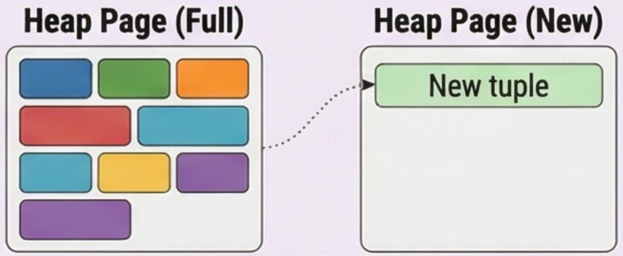
How a HOT chain works: An index scan finds the first tuple. If it's marked "Heap Hot Updated," the scan follows the pointer chain on the heap page to find the current, visible version.

Managing HOT Chains: Pruning & Splitting

Pruning a HOT Chain

Page pruning is smarter with HOT chains. Since intermediate tuples are not referenced by indexes, they can be completely removed. The first tuple in the chain is kept but changed to a "redirect" state, pointing to the current head of the shortened chain.

fillfactor = 75%



Splitting a HOT Chain

If a page becomes full and a new tuple version cannot be stored, the chain is "split." A new tuple is created on a different page, and a new, second entry must be added to the index. This breaks the HOT optimization for that chain.

Use fillfactor to reserve space for HOT updates.

For tables with frequent updates to non-indexed columns, lowering fillfactor (e.g., to 75%) leaves empty space on each page, allowing HOT chains to grow without splitting. The trade-off is a larger overall table size.

PostgreSQL's Cleanup Crew: A Guide to VACUUM & AUTOVACUUM

The Problem: "Dead Tuples" in PostgreSQL

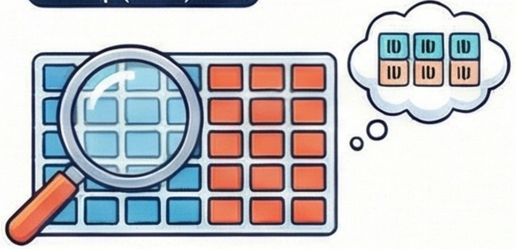
UPDATE and DELETE commands create "dead toplex", instead of everwriting or romoving deta, PeelgroDD), insrbs old row versions as Inactiva, leaving them on disk to ensure transaction visibility.

Read tuples lead to "table bloat". Over tnno, these unuced rows accumulate, consuming disk space and potentially slowing down table scans.

VACUUM is the solution. It's a cleanup process that removes dead tuples and makes the space they occupted available for rouse.

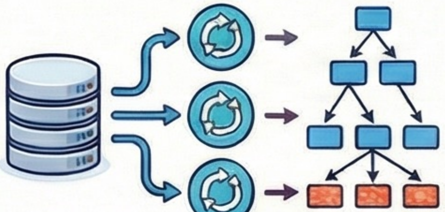
The 4 Stages of a Manual VACUUM

1. Heap (Table) Scan



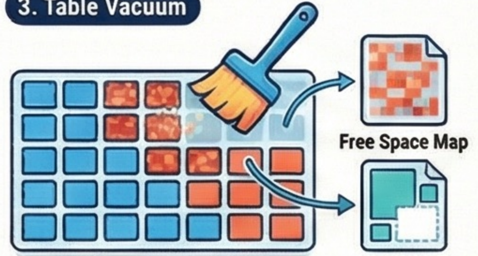
Scans the table to find dead tuples, using the Vixibility Map to skip pages known to have no dead tuples. Their IDs are collected in memory (maintenance_work_mem).

2. Index Vacuum



Scans all of the table's indexes to find and remove entries that point to the collected dead tuples. This can run in parallel for multiple indexes.

3. Table Vacuum



Re-scans the table to remove the actual dead tuple versions, now that they are no longer referenced by any index. Updates the Free Space Map.

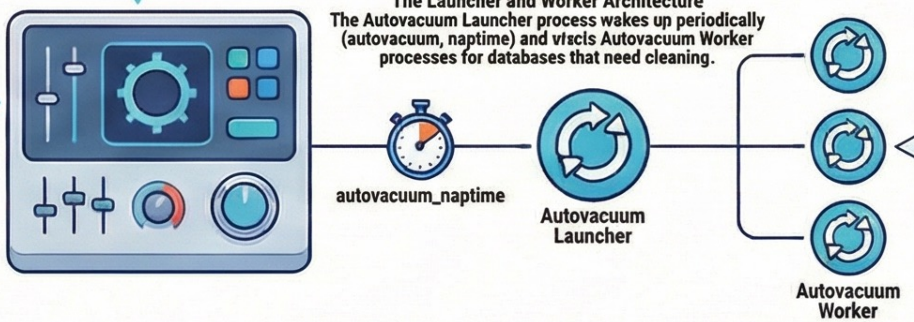
4. Table Truncation

If enough empty pages exist at the end of the table file, this final step may lock the table briefly to return the empty space to the operating system.

The Automation Engine: How Autovacuum Works

Autovacuum solves the scheduling problem. Running **VACUUM** manually is inefficient, tuo often wastes resources, too rarely causes bloat. Autovacuum automates this based on actual table activity.

The Launcher and Worker Architecture
The Autovacuum Launcher process wakes up periodically (autovacuum_naptime) and vtscis Autovacuum Worker processes for databases that need cleaning.



Workers bandle tables coqueniiullc. A single worker processes one table at a time within a database. The total monher of concenrroot workers is limited by autovacuum_max_workers (default: 3).

Autovacuum & Autoanalyze Trigger Conditions

Autovacuum is triggered by thresholds. Autovacuum runs on a table when the number of modified or dead taples exceeds a calculated threshold.

AUTOVACUUM Trigger	Insert-Only VACUUM Trigger	AUTOANALYZE Trigger
VACUUM runs if:	VACUUM runs to update the visibility map if:	ANALYZE (which gathers statistics for the query planner) runs if:
autovacuum_vacuum_threshold (50) > autovacuum_vacuum_scale_factor (50) + (0.2) table_rows * table_rows	autovacuum_vacuum_insert_threshold (1000) > autovacuum_vacuum_insert_scale_factor (1000) + (0.2) table_rows * table_rows	autovacuum_analyze_threshold (50) > autovacuum_analyze_threshold (50) + (0.1) table_rows * table_rows

Default values are shown in persntoeses. These server-leval settings can be overriılden for individual tables to line tune behavior.

Tuning and Monitoring Vacuum Operations

Manage system load with cost-based thruttting.
Autovacuum pauses periodically to avund accouming too many resourees. It does a act amount of "work" (autovacuum_vacuum_sost_inub), then passes for a doration (autovacuum_vacuum_scat_detey).

Default autovacuum_vacuum_sost_doloy is sım. 1ˈtls is a significant change from obler version. (which used 20ms) and is better period for modum herdwars.

Monitor active VACUUM jobs with pg_stal_progress_vacuum.
This system view shows the eursot ghees, pages scumed, and number of index scans (index_vacuum_count).

Multiple Index scans indicate insufficient memory.
If index_vacuum_count is greater than 1, it maans maintenscco_uack_ineor was too sniad to hold all dead topic hile, torving multiple, expensive passes over the indoxes.

Log autovacuum activity for long-term analysis.
Set log_autovacuum_win_durction to 0 io log avery autovacuum run, helping you identify tables that are frequently vacaumed or take a long hide to precuss.

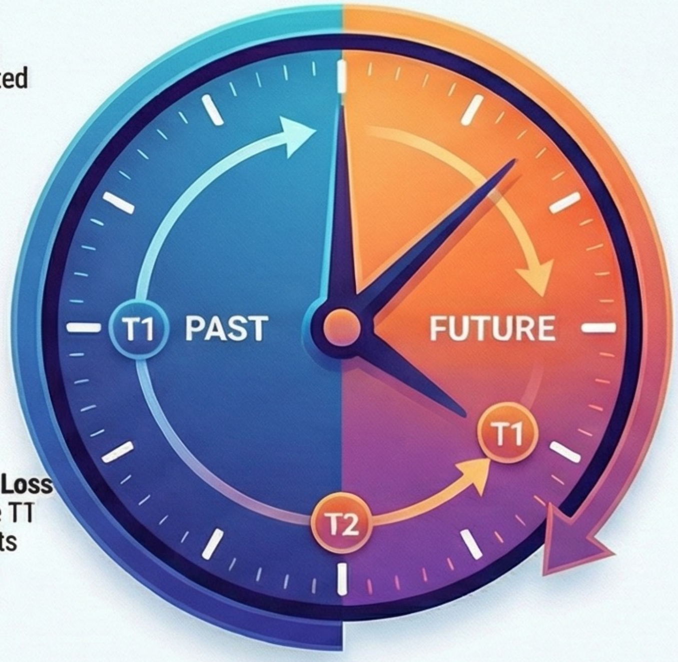
PostgreSQL Freezing: Preventing Transaction ID Wraparound

THE PROBLEM: TXID WRAPAROUND

32-Bit Limit Reached in Weeks: 4 billion Transaction IDs exhausted in about 6 weeks at 1,000 transactions/second.

Age, Not ID, Determines Order: PostgreSQL compares "age" (transactions occurred since) rather than absolute ID number.

The Danger: "Time Travel" Data Loss New transactions incorrectly see TT as being in the "future", making its changes invisible and leading to catastrophic data inconsistency.



THE SOLUTION: TUPLE FREEZING

What is Freezing? VACUUM identifies old tuples and marks them as "frozen", meaning they are universally visible to all transactions, past and future.

Modern Freezing Uses Hiot Bits: Previously changed tuple's `xmtn` to special value (2). Now, sets two "hint bits" in the header, preserving original `xmln` for debugging.

Frozen Tupies are Infinitely Old: A frozen tuple is treated as if its creation transaction is in the distant past for every other transaction, safely removing it from wraparound risk.

HOW FREEZING IS MANAGED AND AUTOMATED

Step 1: Routine Freezing
`vacuum_freeze_min_age`
(Default: 50 million transactions)
Standard VACUUM freezes tuples older than this age on processed pages.

Step 2: Aggressive Freezing
`vacuum_freeze_table_age`
(Default: 150 million transactions)
If table's oldest unfrozen TXID (relfrozenxid) exceeds this, VACUUM scans *all* pages.

Step 3: Forced Autovacuum
`autovacuum_freeze_max_age`
(Default: 200 million transactions)
A safety net. Autovacuum is *forced* to run on a table if it gets too old, even if disabled.

datfrozenxid
'Laggard Table' determines database's freeze age.

Step 4: Emergency Failsafe
`vacuum_fallsafe_age`
(Default: 1.6 billion transactions)
The final defense. High-priority VACUUM runs, skipping non-essential work to freeze tuples as fast as possible to prevent shutdown.

TAKING MANUAL CONTROL OF FREEZING

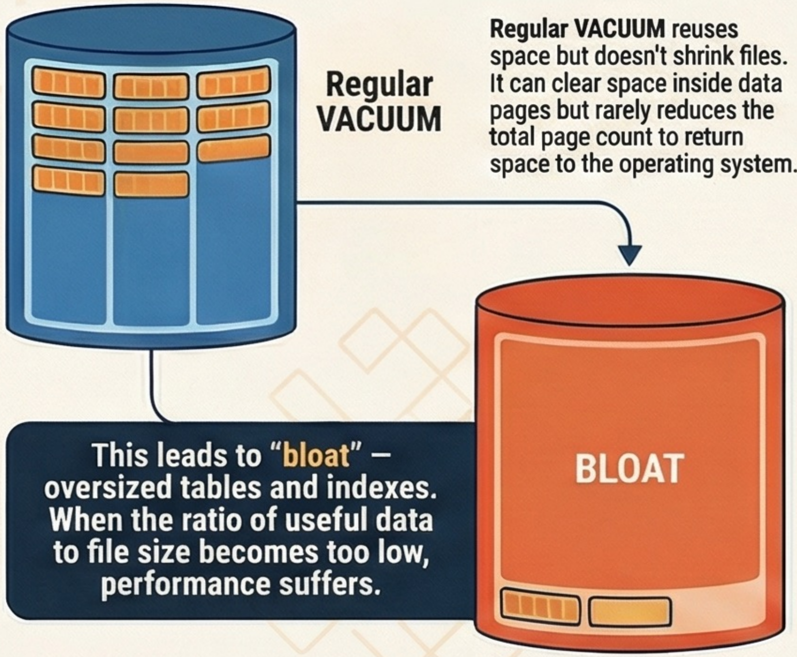
"VACUUM FREEZE": This command performs an aggressive freeze on an entire table immediately, freezing all tuples regardless of their age.

"COPY ... WITH FREEZE": For bulk-loading static data, this command freezes the rows as they are inserted, preventing future VACUUM on them (as long as they don't change).

Ideal for Unchanging Data: Manually freezing is most useful for tables that are loaded once and rarely or never updated, avoiding unnecessary vacuum overhead.

Taming the Bloat: A Guide to PostgreSQL Table Reorganization

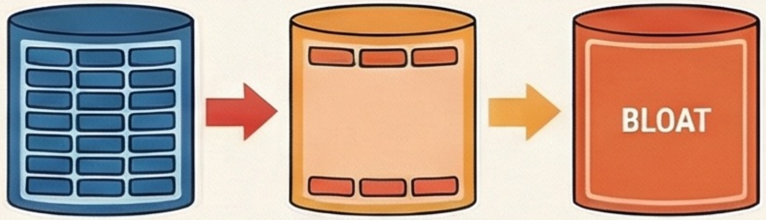
The Problem: Why Regular VACUUM Isn't Enough



Regular VACUUM reuses space but doesn't shrink files. It can clear space inside data pages but rarely reduces the total page count to return space to the operating system.

This leads to "**bloat**" – oversized tables and indexes. When the ratio of useful data to file size becomes too low, performance suffers.

A Practical Demonstration of Bloat



Start with a healthy table.
500,000 rows.
Table Size: 67 MB
Data Density (tuple_percent): 91.33%.

Delete 90% of the date.
450,000 rows removed.

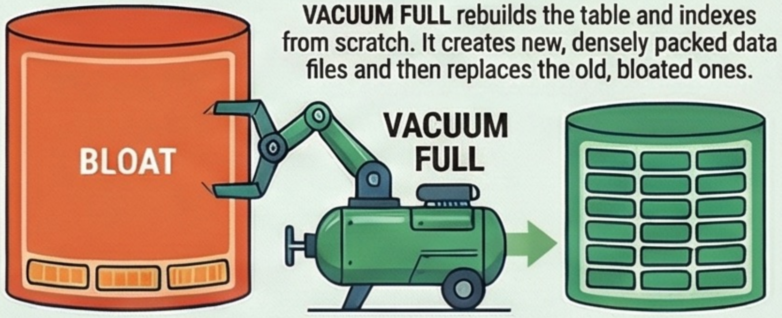
File Size Unchanged: 67 MB
Despite measive delation, the table stiid occupies the same disk space.

Run a standard VACUUM.

Data Density Plummets: 9.13%
Over 90% of the file is now free space.

The Solution: Rebuilding with VACUUM FULL

VACUUM FULL rebuilds the table and indexes from scratch. It creates new, densely packed data files and then replaces the old, bloated ones.



Before
Table Size: 67 MB | Index Size: 109 MB

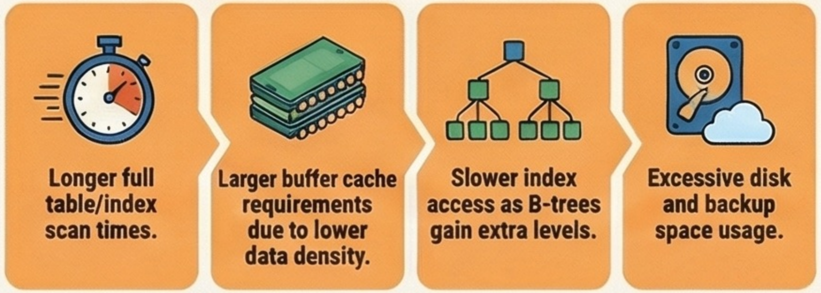
Drastic Size Reduction:
The table size chrinks significantly, and index size is reduced.

After
Table Size: 67 MB | Index Size: 6.5 MB

Data Density Restored:
tuple_percent restored to 91.23%, and index evg_leaf_density to 91.08%.

Major Drawbock: Exclusive Table Lock. VACUUM FULL blocks all read and write access while it runs, making it unsuitable for high-availability systems.

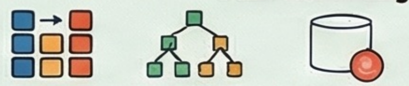
The Consequences of Bloat



- Longer full table/index scan times.
- Larger buffer cache requirements due to lower data density.
- Slower index access as B-trees gain extra levels.
- Excessive disk and backup space usage.

Alternatives to VACUUM FULL

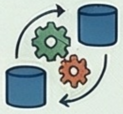
Other Rebuilding Commands



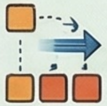
Rebuilds the table while physically reordering rows based on a specified iniss main table data. Quickly deletes all rows by creating a new empty file, which is much faster than DELETE.

For Minimal Downtime: Use Extensions

Tools exist to rebuild tables without a prolonged exclusive lock.



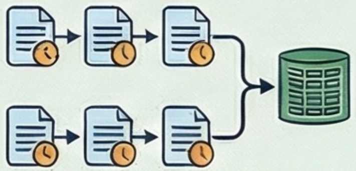
pg_repack
Creates a new table in the background, tracks changes with triggers, and their swaps, requiring only firrel locks.



pgcompacttable
Gradoally moves rows to the beginning through small updates and vacumns. Slower but requires no extra disk spacs.

Preventing Bloat Before It Starts

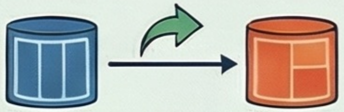
Problem: Long-Running Transactions



Mitigate by separating analytical queries (ULAP) outo read replicas or using timeouts like **idle_in_transaction_session_timeout.**

Problem: Mass Data Updates

Updating every row in a large table at oncs can double its size.



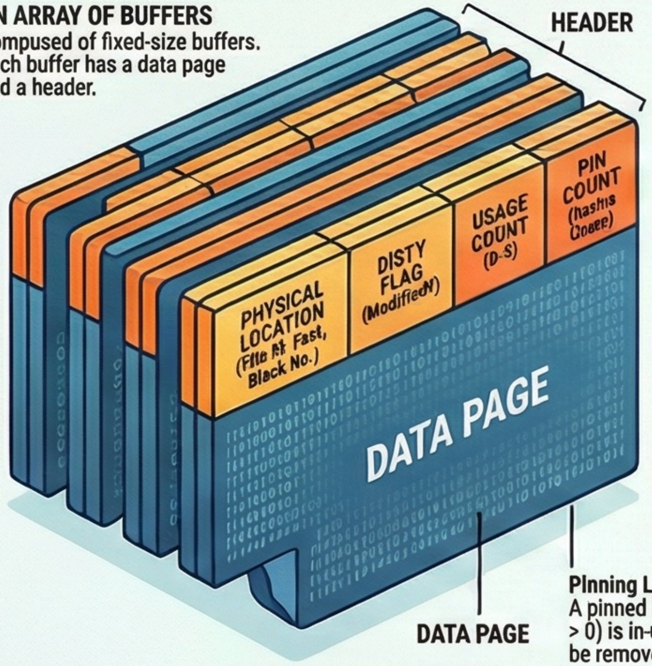
Solution: Process in Batches. Instead of one large **UPDATE**, process rows in smaller batches. An intermediate **VACUUM** can then clear old rows, allowing new rows to reuse the freed space, keeping the table size stable.

NotebookLM

PostgreSQL's Brain:
A Deep Dive into the Buffer Cache

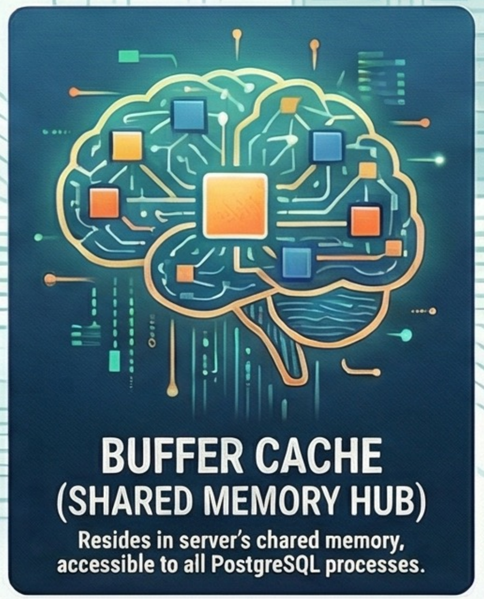
ANATOMY OF THE BUFFER CACHE

AN ARRAY OF BUFFERS
Compused of fixed-size buffers. Each buffer has a data page and a header.



THE HEADER STORES METADATA
Each header holds vital information about its data page.

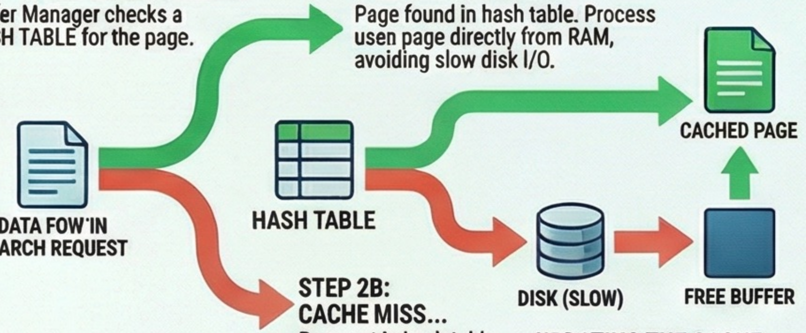
Plnning Locks the Buffer:
A pinned buffer join count > 0) is in-use and cannot be removed.



THE PAGE REQUEST LIFECYCLE: HIT OR MISS?

STEP 1: THE SEARCH BEGINS
Buffer Manager checks a HASH TABLE for the page.

STEP 2A: CACHE HIT!
Page found in hash table. Process usen page directly from RAM, avoiding slow disk I/O.



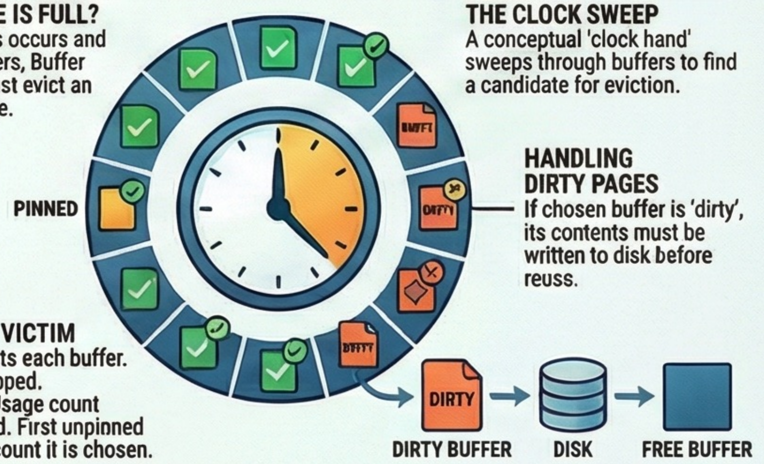
STEP 2B: CACHE MISS...
Page nut in hash table. System must read from disk into a free buffer.

UPDATING THE CACHE
After a miss, page is loaded, metadata updated in header, reference added to hash table, usage count set to 1.

MAKING SPACE: THE EVICTION STRATEGY

WHAT HAPPENS WHEN THE CACHE IS FULL?
When a miss occurs and no free buffers, Buffer Manager must evict an existing page.

THE CLOCK SWEEP
A conceptual 'clock hand' sweeps through buffers to find a candidate for eviction.



HANDLING DIRTY PAGES
If chosen buffer is 'dirty', its contents must be written to disk before reuss.

FINDING A VICTIM
Hand inspects each buffer. Pinned? Skipped. Unpinned? Usage count decremented. First unpinned with usage count it is chosen.

HANDLING BULK OPERATIONS

THE THREAT OF CACHE POLLUTION
Large operations can flood the buffer cache, pushing out hot pages.

THE BUFFER RING SOLUTION
PostgroSQL uses small, dedicated 'buffer rings' for bulk operations to contain exiction.

THREE EVICTION STRATEGIES:
Different strategies depending on the operation type.

TUNING AND ADMINISTRATION

KEY FINDING SIZING YOUR CACHE (shared_buffers)
Default size (125MB) is too low. Start with 25% of total RAM.

HOW TO MONITOR WITH pg_buffercache
Inspect cache contents to see cached relations and 'hot' pages (based on usagecount).

HOW TO PRE-WARM THE CACHE WITH pg_prewarе
Load important tables into cache after restart or save/restore cache state.

THE EXCEPTION: LOCAL CACHE FOR TEMPORARY TABLES

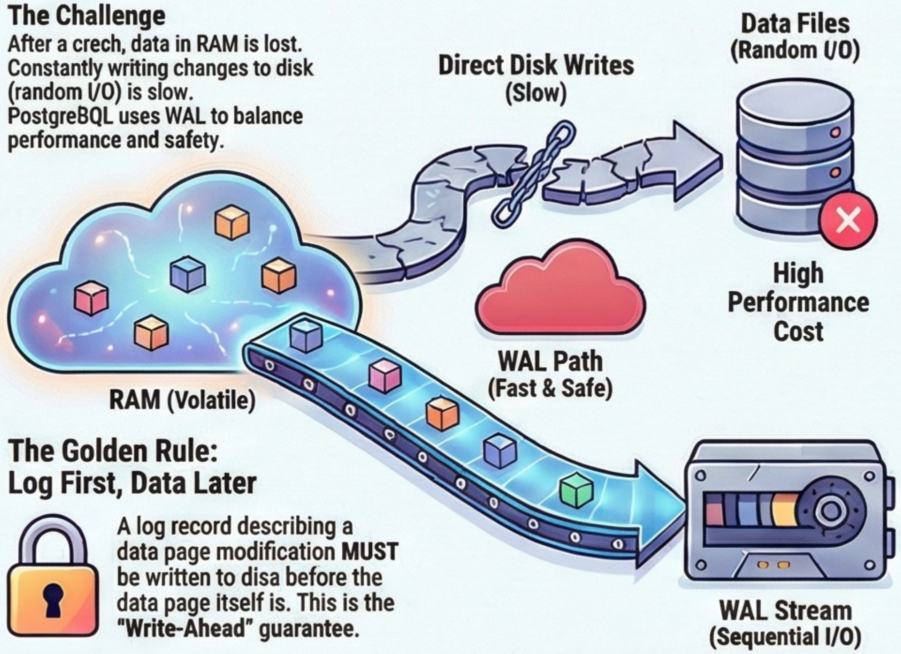
TEMPORARY TABLES GET THEIR OWN CACHE
Temporary data is session-private, uses a simpler, local cache instead of the shared buffer cache.

SIMPLER AND MORE EFFICIENT
Local cache doesn't need complex loeking. Size controlled by temp,,buffers parameter (default 8MB).

A Visual Guide to PostgreSQL's Write-Ahead Log (WAL)

1. The "Why": Durability vs. Performance

The Challenge
After a crech, data in RAM is lost. Constantly writing changes to disk (random I/O) is slow. PostgreBQL uses WAL to balance performance and safety.



The Golden Rule: Log First, Data Later

A log record describing a data page modification **MUST** be written to disa before the data page itself is. This is the "Write-Ahead" guarantee.

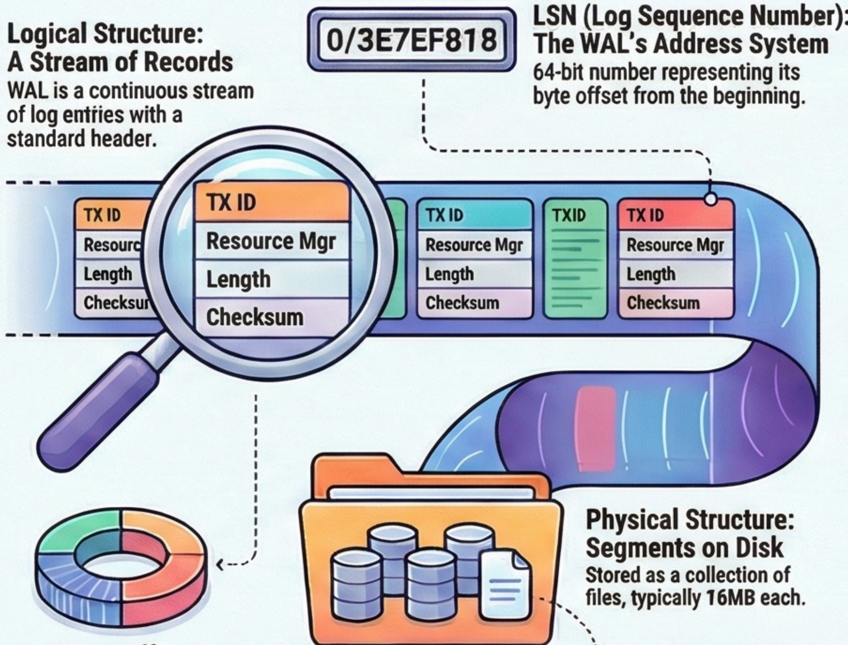
What Gats Logged?
- Page modifications in buffer cache
- Transaction commits/rollbacks
- File operations

What is NOT Logged?
- Operations on UNLDGGED & temporary tables

2. The "What": Anatomy of the WAL

Logical Structure: A Stream of Records
WAL is a continuous stream of log entries with a standard header.

LSN (Log Sequence Number): The WAL's Address System
64-bit number representing its byte offset from the beginning.



WAL Buffers (Shared Memory)
Cache for now WAL records before Rushing to disk.

PGDATA/pg_wal

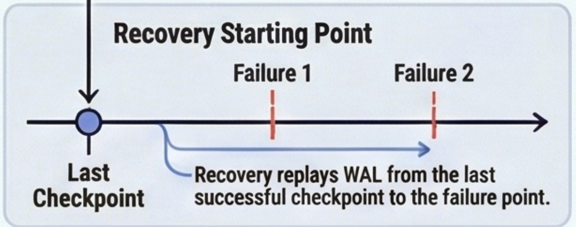
Physical Structure: Segments on Disk
Stored as a collection of files, typically 16MB each.

Filename Example:
000000010000000000000030
Timeline ID + Log Sequence Number.

3. The "How": The Checkpoint Process

Checkpoint: Safe Starting Point
A point in the WAL sequence where all modified data pages are flushed to disk.

Why Necessary?
Create a known eafe point for recovery, allowing oid WAL files to be recycled.



The 3 Phases of a Checkpoint (checkpointer process)

1. **Start** — Identify doty pages.
2. **Execution** — Write doty pages to dish over time.
3. **Completion** — Write checkpoint record, update pg_control.

pg_control → Points to REDO location (start of recovery).

4. Putting It All Together: Crash Recovery

Crash Detected

1. **Read pg_control** — If state is "In production", trigger recovery. →
2. **Find Starting Point** — Begin from "REDO location" in pg_control (last checkpoint). →
3. **Replay the Log** — Read WAL forward, apply changes if data page is older.

No lback Needed
Roll forward. Uncommitted ractions are treated as aborted.

4. **Recovery Complete** — New checkpoint performed, database ready.

READY

5. Tuning and Monitoring WAL

Balancing Checkpoint Triggers
Triggers: checkpoint_timeout (time) OR mes_wal_size (sisc).
Gost: Most checkpoints time-based.

Warning Signs
If size-based checkpoints are much more frequent, max_wal_size may be too low.

Role of Background Writer (bgwriter)
Preactively writes pages to disk, reducing checkpointer work.
Key Metric: buffers_backend (in pg_stat_bgwriter) should be LOW.

Key Configuration Parameters

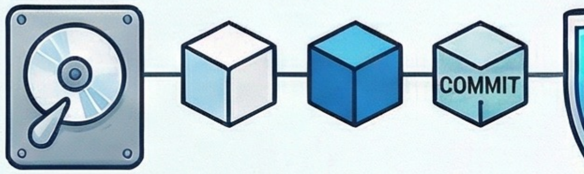
Parameter	Default	What it Does	Value
checkpoint_timeout	5 min	Max time between automatic checkpoints	
max_wal_size	1 GB	WAL size triggering a checkpoint	
checkpoint_completion_target	0.9	Spreads checkpoint I/O over time	
min_wal_size	80 MB	Min WAL size to keep for reuse	

NotebookLM

The Performance vs. Durability Trade-off: Inside PostgreSQL's WAL Mode

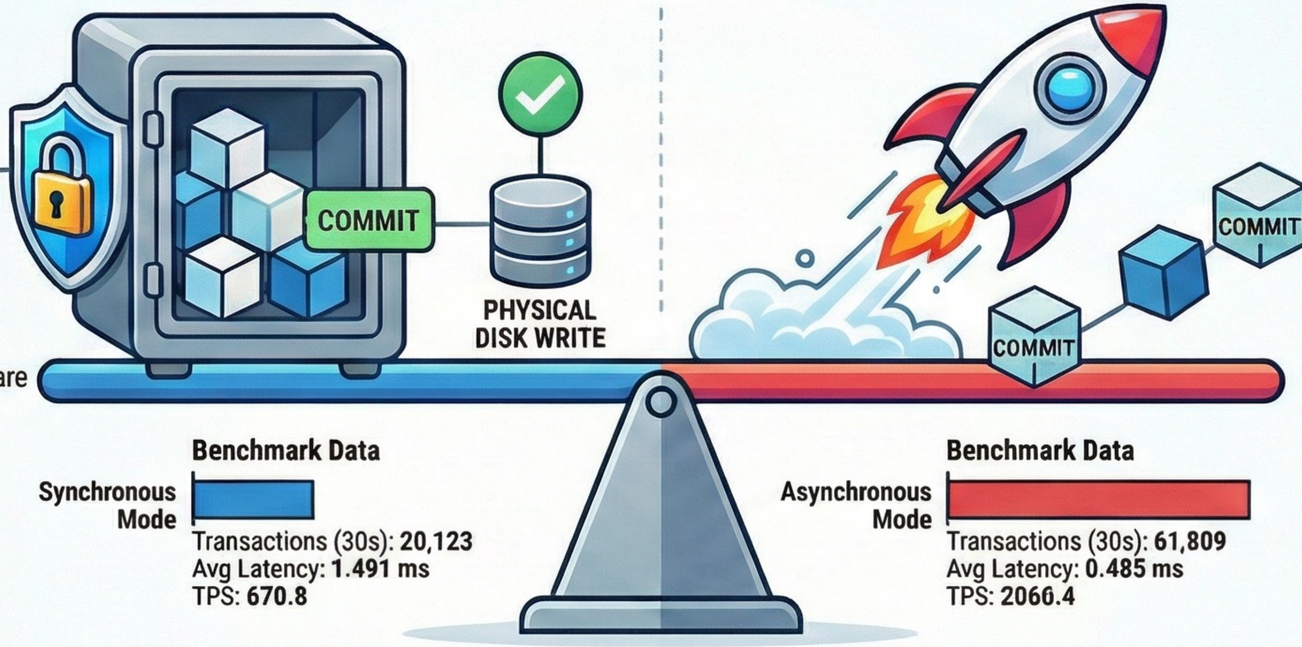
Synchronous Commit: The Safety-First Approach

Transaction only 'committed' after WAL records physically written to disk.
(Default: synchronous_commit = on)



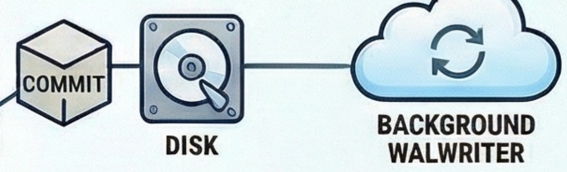
Pro: Maximum Reliability
Ensures ACID durability requirements are met. Once a commit is acknowledged, the data is safe from crashes.

Con: Slower Performance
Waiting for disk I/O increases latency and reduces throughput.



Asynchronous Commit: The Speed-Focused Alternative

Transaction confirmed immediately; WAL records written to disk in background.
(synchronous_commit = off)



Pro: Higher Throughput & Lower Latency
Eliminates disk I/O wait, making commite significantly faster.

Con: Risk of Data Loss
...recently committed transactions can be lost (a window of up to 0.6 seconds by default).

Ensuring Fault Tolerance

Challenge 1: Non-Atomic Writes
Database page (8KB) written in smaller blocks (4KB). Crash can leave corrupted, partial page.

Solution: Full Page Writes (FPI)
PostgreSQL writes a full copy (FPI) to WAL on first modification after checkpoint for recovery.

FPIs increase WAL size (e.g., 71.5% of data). Enable 'wal_compression = on' to reduce size (e.g., 29 MB to 10 MB).

Challenge 2: Data Corruption
Hardware failures can silently corrupt data in memory, transfer, or on disk, spreading to backups.

Solution: Checksums
Enable 'data_checksums' to verify page checksum on read. WAL records always protected by checksums.

Understanding WAL Levels

Level 1: Minimal
(wal_level = minimal)
Logs essential crash recovery info. Skips bulk operations to save space. No backups/ replication.

Level 2: Replica
This is the default level.
Logs enough for Point-in-Time Recovery and Physical Streaming Replication. All data changes logged.

Level 3: Logical
(wal_level = logical)
Includes Replica info plus data for Logical Decoding. Required for Logical Replication to other systems.

Understanding Database Locking in PostgreSQL

Database locking is a fundamental mechanism for managing concurrency, preventing data corruption by controlling simultaneous access to shared resources. PostgreSQL employs a sophisticated system to balance performance and data integrity.

The Fundamentals of Locking

What is a Lock?
A lock is a mechanism that controls concurrent access to a shared resource, ensuring that multiple processes don't interfere with each other.

The Lock Lifecycle

1. Acquire a lock on a resource
2. Perform its operation
3. Release the lock so other processes can use the resource.

The Trade-off: Finer-grained locks increase concurrey but also increase the number of locks to manage. Coarse-grained locks are simpler but limit concurrency.

Common Heaverlyweight Lock Types

- **relation**: Table-level locks
- **tuple**: Row-level locks
- **transactionid**: Locks on a transaction itself
- **page**: Locks on data pages (used by some indexes)
- **advisory**: User-managed locks
- **object**: Locks on non table database objects

No Concurrency = No Locks
If a resource is not accessed simultaneously by rrocesse it doesn't require a lock.

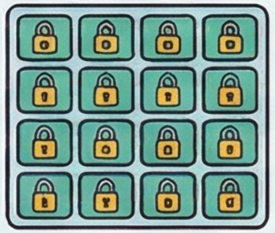
Key Characteristics of Locks

Granularity: The Scope of the Lock

Coarse-grained

A table-level lock is coarse, preventing all concurrent access to that table.

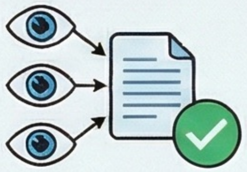
Fine-grained



A row-level lock is fine, allowing processes to work on different rows of the same table simultaneously.

Lock Modes: The Type of Access

Shared Mode



Allows multiple processes to reed a resource simultaneously.

Exclusive Mode

Prevents any other process from accessing the resource; used for writing.

Basic Compatibility (Shared vs. Exclusive)

	Shared	Exclusive
Shared	✓	✗
Exclusive	✓	✗
Exclusive	✗	✗

Classifying Locks by Duration

Long-term Locks
Held for a potentially long time, often until a transaction ends. Protects resources like tables and rows. Includes advanced features like wait queues and deadlock detection.

Short-term Locks
Acquired for very brief periods (microseconds) to protect data structures in shared memory. Managed automatically with simple infrastructure.

A Closer Look at PostgreSQL's Heavyweight Locks

What are Heavyweight Locks?
These are the long-term, object-level locks in PostgreSQL, visible in 'pg_locks' view, managed in a shared memory pool.

Common Heavyweight Lock Types ("locktype")

- **relation:** Table-level locks
- **tuple:** Row-level locks
- **transactionid:** Locks on a transactor itself
- **page:** Locks on data pages
- **advisory:** User-managed locks
- **object:** Locks on non-table objects

	AS (Aocess Shars)	RS (Row Xtore)	SUE (Sharo Update Exclusive)	E (Share Row /\6)	AE (Eeilsatle)
S		✓	✗	✓	✗
SRE	✓		✗	✓	✗
S	✗	✓		✓	✗
E	✗	✓	✓		✗
AE	✗	✗	✗	✗	

Table-Level Lock Modes & Compatibility

8 Modes for Maximum Concurrency

PostgraSQL provides eight table level lock modes to allow the maximum number of operations to run in parallel without conflict.

Most Permissive: Access Share
Used by 'SELECT' queries. Compatible with all modes except the most restrictive, allowing reads alonggside alnot t any other operation.

Most Restrictive: Accease Exclusive
Used by crrommonds like 'DROP TABLE'E' or 'VACUUM FULL'. Incompatible with all other lock msiodes, ensuring no other process can access the table.

The Wait Queue in Action

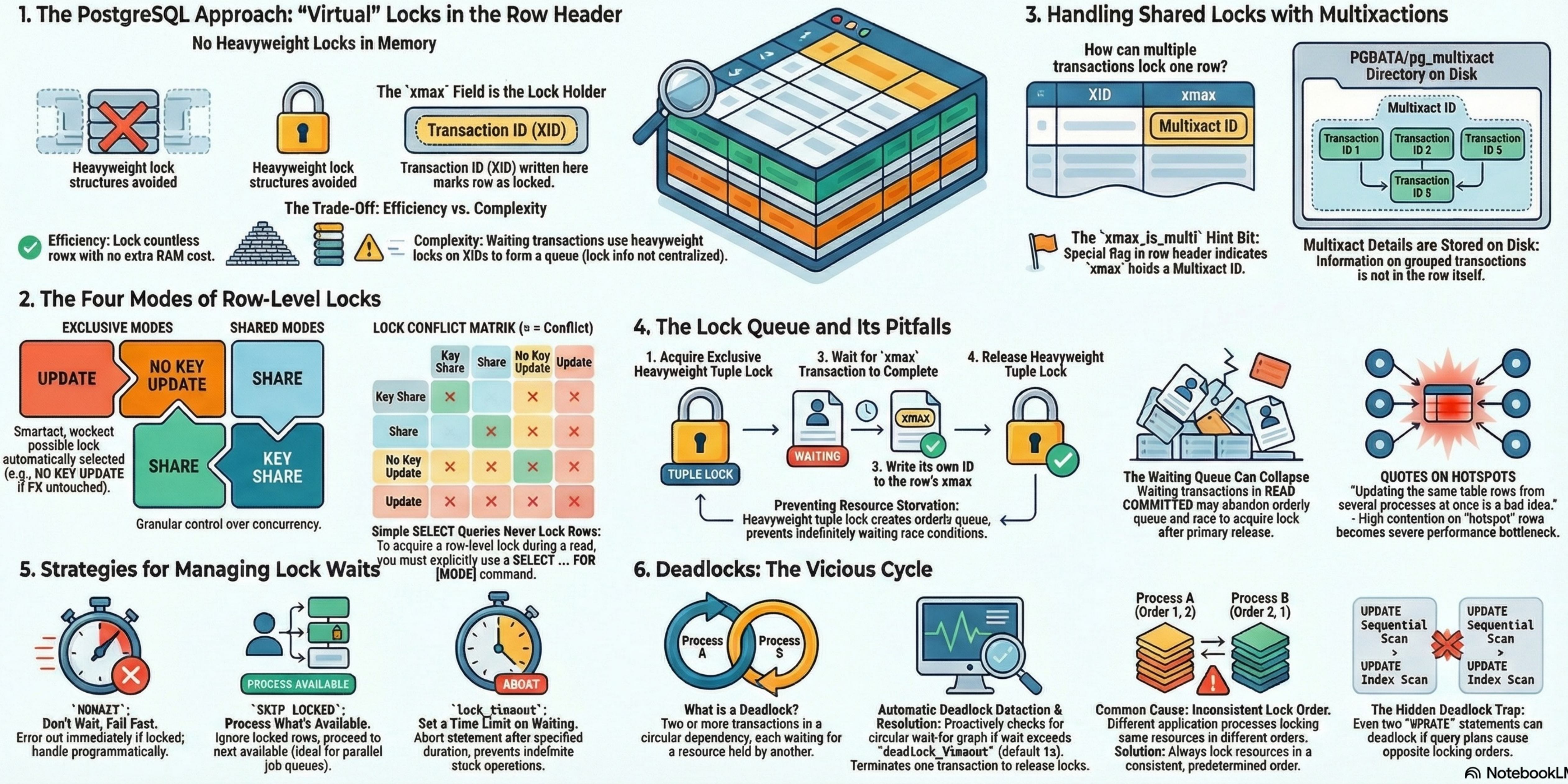
What is a Wait Queue?
When a process tries to acquire a lock that conflicts with an existing lock, it enters a "firet-in, first-out" wait queue until the resource is freed.

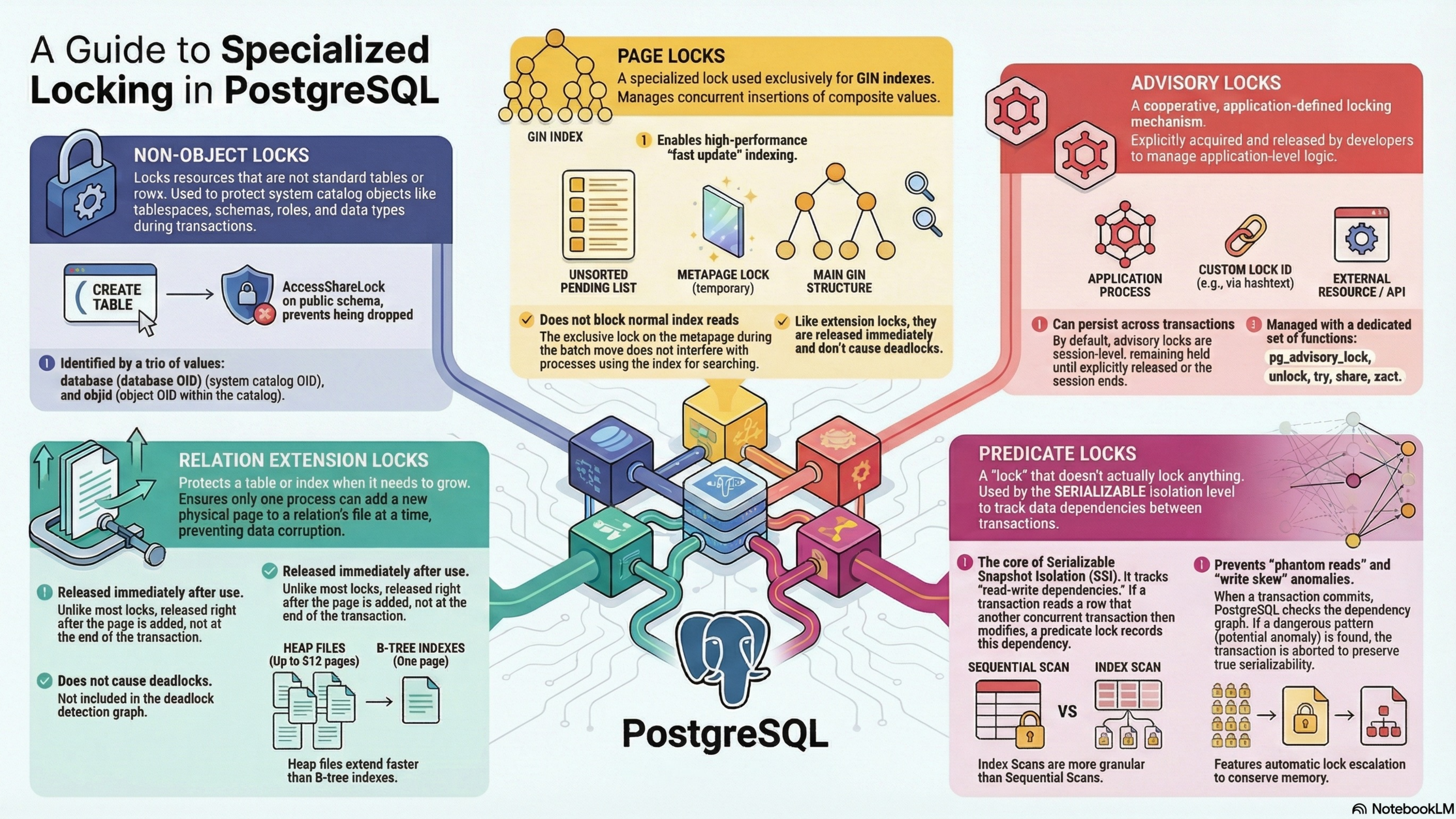
Table-Level Lock Compatibility Matrix

	AS	RS	RE	SRE	S	X	AE
AS		✓	✓	✗	✗	✓	✗
RS	✓	✓		✗	✗	✓	✗
RE	✗		✓	✓		✓	✗
SRE	✗	✓		✓	✓	✓	✗
S	✗	✓	✓		✓	✓	✗
X	✗	✗	✓	✗	✓	✓	
AE	✗	✗	✓	✗	✗		✓

Deadlock Detection
If two or more transactions waiting for each other in a circular chain, PostgreSQL automatically detects this "deadlock" and terminates one of the transactions.

A Deep Dive into PostgreSQL Row-Level Locking




A Guide to Specialized Locking in PostgreSQL
NON-OBJECT LOCKS
Locks resources that are not standard tables or rowx. Used to protect system catalog objects like tablespaces, schemas, roles, and data types during transactions.
CREATE TABLE
AccessShareLock on public schema, prevents heing dropped
Identified by a trio of values: database (database OID) (system catalog OID), and objid (object OID within the catalog).
PAGE LOCKS
A specialized lock used exclusively for GIN indexes. Manages concurrent insertions of composite values.
GIN INDEX
1 Enables high-performance "fast update" indexing.
UNSORTED PENDING LIST
METAPAGE LOCK (temporary)
MAIN GIN STRUCTURE
Does not block normal index reads
The exclusive lock on the metapage during the batch move does not interfere with processes using the index for searching.
Like extension locks, they are released immediately and don't cause deadlocks.
ADVISORY LOCKS
A cooperative, application-defined locking mechanism.
Explicitly acquired and released by developers to manage application-level logic.
APPLICATION PROCESS
CUSTOM LOCK ID (e.g., via hashtext)
EXTERNAL RESOURCE / API
Can persist across transactions
By default, advisory locks are session-level, remaining held until explicitly released or the session ends.
Managed with a dedicated set of functions:
pg_advisory_lock,
unlock, try, share, zact.
RELATION EXTENSION LOCKS
Protects a table or index when it needs to grow. Ensures only one process can add a new physical page to a relation's file at a time, preventing data corruption.
Released immediately after use.
Unlike most locks, released right after the page is added, not at the end of the transaction.
Released immediately after use.
Unlike most locks, released right after the page is added, not at the end of the transaction.
Does not cause deadlocks.
Not included in the deadlock detection graph.
HEAP FILES (Up to $12 pages)
B-TREE INDEXES (One page)
Heap files extend faster than B-tree indexes.
PostgreSQL
PREDICATE LOCKS
A "lock" that doesn't actually lock anything. Used by the SERIALIZABLE isolation level to track data dependencies between transactions.
The core of Serializable Snapshot Isolation (SSI). It tracks "read-write dependencies." If a transaction reads a row that another concurrent transaction then modifies, a predicate lock records this dependency.
Prevents "phantom reads" and "write skew" anomalies.
When a transaction commits, PostgreSQL checks the dependency graph. If a dangerous pattern (potential anomaly) is found, the transaction is aborted to preserve true serializability.
SEQUENTIAL SCAN
VS
INDEX SCAN
Index Scans are more granular than Sequential Scans.
Features automatic lock escalation to conserve memory.
NotebookLM

A Deep Dive into PostgreSQL's Memory Locks

The Locking Toolkit: Spinlock vs. LWLock

Understanding the Core Mechanisms

Spinlock: The Sprinter

- **Type:** Exclusive lock using atomic CPU commands for extremely short durations (a few CPU cycles).
- **How It Handles Contention:** “Busy-Waiting”. Processes repeatedly check in a tight loop (“spin”) until free. Efficient only for rare, brief contention.
- **Limitations:** Exclusive mode only. No built-in deadlock detection or monitoring instrumentation.

LWLock (Lightweight Lock): The Marathoner

- **Type:** Acquired for longer periods to manage data structures, sometimes spanning I/O operations.
- **How It Handles Contention:** Shared & Exclusive Access. Supports multiple simultaneous readers and single writer, offering flexibility.
- **Limitations:** LWLocks lack deadlock detection but include instrumentation for observability. No queue; access is relatively random.

Locks in Action: Critical Shared Memory Structures

The Buffer Cache

BufferMapping LWLock (x128) ↔ Buffer Cache Hash Table

To find a buffer, a process must acquire one of 128 LWLocks protecting the buffer hash table, preventing bottlenecks.

header

Buffer Pool

- **buffer header Spinlock:** Fast, exclusive access to buffer metadata.
- **BufferContent Lock:** Acquired on buffer header to read/write a page inside.
- **BufferIO Wait Signal:** Flag set during disk I/O, telling others to wait.
- **buffer strategy Spinlock:** Protects free buffer management and eviction (“clock hand”).

The WAL Buffer

WAL Cache Hash Table

WALBufMapping LWLock

A single LWLock protects the entire WAL cache hash table, as access is smaller and more sequential than the main buffer cache.

WAL Pages

- **WALWrite LWLock:** Guarantees only one process writes WAL records to disk at a time.
- **insert position Spinlock:** Acquired to reserve space sequentially in a WAL page.
- **WALInsert LWLock (x8):** After reserving space, multiple processes write data concurrently by acquiring one of eight LWLocks.

Monitoring Lock Waits & Performance

Turning Abstract Concepts into Actionable Insights

Why Monitor Waits?

Locks can cause waits, creating performance bottlenecks. Tracking “wait events” is crucial.

Step 1: Real-Time Check with `pg_stat_activity`. Live snapshot of current `wait_event_type` and `wait_event` for active processes.

Step 2: Historical Analysis Since `pg_stat_activity` is not cumulative, an extension like `pg_wait_sampling` is needed to collect wait statistics over time for a complete picture.

Common Wait Event Categories & Analysis

- LWLock
- Lock (heavy)
- BufferPin
- IO (disk)
- Client (network)
- IPC (inter-process)

pgbench

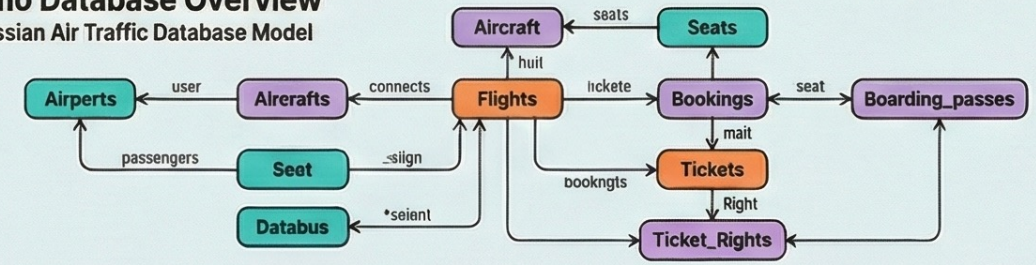
Sample Analysis: `pg_wait_sampling_profile` might reveal frequent `IO` events like `WAL-Tyoc` and `WALWrite`, indicating a disk I/O bottleneck.

NotebookLM

The Life of a PostgreSQL Query: From Text to Result

Demo Database Overview

A Russian Air Traffic Database Model



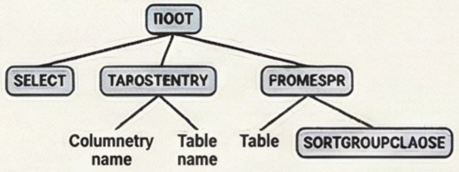
Step 1: Parsing

The server reads the raw SQL text to understand its structure.

Raw SQL

Lexer & Parser

Simple Parse Tree

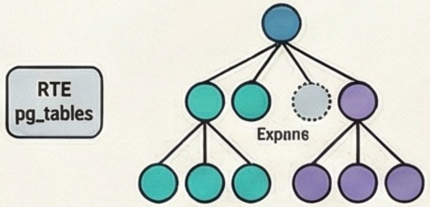


A "Lexer" splits the text into tokeos (seowards, identifiers). A "Parser" checks if these tokens follow SQL grammar rules and builds the AST.

The system checks the AST against the database catalog. The system checks the AST against the database data.

Step 2: Transformation (Rewrite)

The initial parse tree is modified and expanded.



QUERY Sub-tree

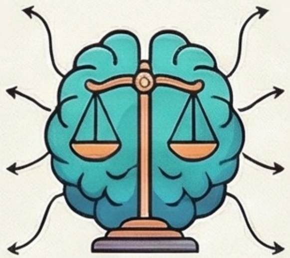
Views are Replaced with Subqueries: The primary use is to replace any reference to a view (llse pg_tables) with the underlying SELECT query, and oorers are expandies.

Row-Level Security policies: Security rules are injected into the query tree. and conno-pnan mat are imjected into the outonle query tree.

Step 3: Planning (Optimization)

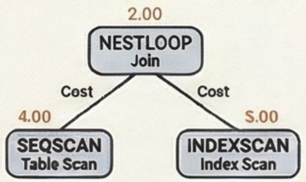
The Planner finds the most efficient execution path.

SQL is declarative... The planner, a cost-based optimizer, evaluates many potential plans to find the "cheapeat" one.



What is a "Plan"?
A plan is a tree of physical operations (e.g., table scan, ndex scan, index scan, nested loop join) that will retnove the data.

Sample Plan Tree

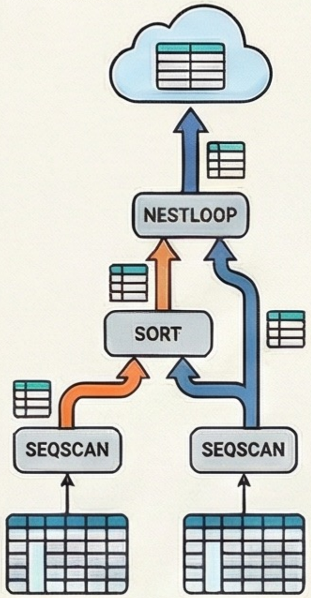


Cost is based on Statistics: To estimate costs, the planner relles on statistics about the data (table size, data distribution).

Step 4: Execution

The Executor runs the chosen plan.

The evecutor works its way through the plan tree, with each parent node "puiling" rows from its child nodes.



Data Flows Up the Tree: Leaf nodes (like SEQSCAN) read from tables. The rows are passed up... to the client.

Memory Usage: Operations that need to accumulate data (like SORT) use memory allocated from work_mem. If exceeded, data is spilled to temporary disk files.

Simple vs. Extended Query Protocols

Simple Protocol

One Request, One Response

Full Result Set

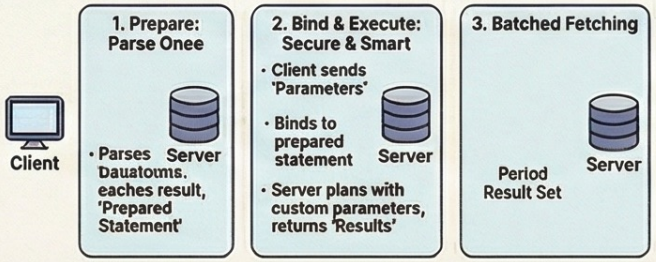
Disadvantages:

Inefficient for Repetitive Queries: The same query sent multiple times must be parsed, transformed, and planned every single time.

All-or-Nothing Results: The client must wait for the full result set, which cav fie slow for tiage queries.

Extended Protocol

A More Oranular Approach

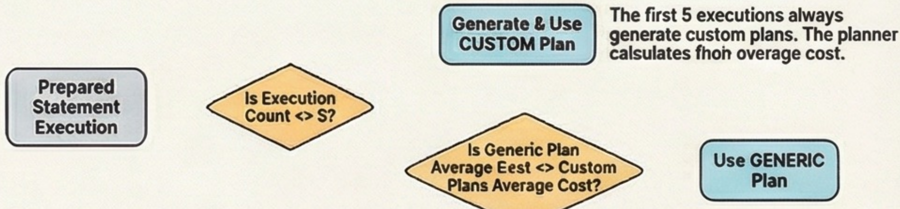


Advantages:

1. **Prepare: Parse Onee:** The query is pulsed and transformed once, and the result is sached. Transmlon, the prepared statement.... asolds redondant parsing.
2. **Bind & Execute: Secure & Smart:** Parameter values are sont separately... presents SQL injection and allows the planner to create a custom plan based on the specific values.
3. **Batched Fetching:** The client can fetch results in smaller batches, instead of prepanuš results... instead of waiting for the entire set...

Custom vs. Generic Plans in the Extended Protocol

The Planner's Dilemma: Re-plan or Reuse?



Why It Matters: A generic plan might be faster on average but can be very inefficient for specific parameter values, and roues... controlled with the plan_cachs_mode setting.

The PostgreSQL Planner's Secret Weapon: A Guide to Query Statistics

From basic table counts to advanced multivariate analysis, here's how PostgreSQL estimates costs and chooses efficient execution plans.

THE FOUNDATION: RELATION (TABLE) LEVEL STATISTICS

Basic counts for every table.

PostgreSQL stores high-level statistics for each table in the **pg_class** system catalog, providing the planner with a starting point.

KEY METRICS: reltuples & relpages

reltuples (Row Count)
Number of rows; default estimate for queries without filters.

relpages (Table Size in 8KB Pages)
Total table size in pages.

STATISTICS COLLECTION

How are they collected?

Statistics are gathered during manual or automatic **ANALYZE** operations, as well as other maintenance tasks like **VACUUM**, **CLUSTER**, and **CREATE INDEX**.

KEY FINDING

The planner is smarter than its data.

If actual file size is larger than relpages suggests, the planner adjusts its row estimate upwards.

EXAMPLE

Before ANALYZE
Planner assumes default (e.g., 410 rows, reltuples = -1).

After ANALYZE
Estimate becomes accurate based on gathered data.

DEEPER DIVE: COLUMN-LEVEL STATISTICS

Understanding the data inside each column.

Stored in **pg_statistic**, these stats describe data distribution, crucial for estimating **WHERE** clause selectivity.

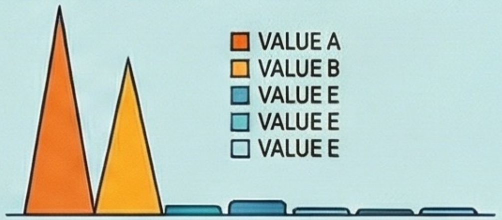
null_frac: The frequency of NULLs

Fraction of NULL entries, used to estimate IS NULL and IB NOT NULL conditions.

n_distinct: The number of unique values

For uniform data, planner divides total rows by n_distinct for equality checks.

VISUAL CONCEPT
For non-uniform data: Most Common Values (MCV)



Stores exact frequencies of popular values for highly accurate estimates on specific value filters.

VISUAL CONCEPT
For many unique values: Histograms

Groups remaining data into buckets with roughly equal items, storing only boundaries for range queries (<, ≠).

ADVANCED TOOLS: EXPRESSION & MULTIVARIATE STATISTICS

PROBLEM

```
WHERE extrect(mouth FROM date_col) = 1
```

Planner can't guess function results, uses inaccurate fixed guess (e.g., 0.5%).

SOLUTION

Planner can't guess function results.
Planner can't guess function results, uses inaccurate fixed: (e.g., 0.5%).

Solution 1: Extended Statistics on Expressions
Use CREATE STATISTICS to collect detailed stats (n_distinct, MCV) on expression results.

Solution 2: Indexing an Expression
Creating an index on an expression automatically generates and stores statistics.

PROBLEM

flight_nq — deporture_sirport

SOLUTION

Columns are not always independent.
Planner assumes independence, leading to severe underestimates for correlated columns.

SOLUTION

Solution: Multivariate Statistics
Use CREATE STATISTICS to analyze relationships, including Functional Dependencies, Ħ-Distinct Combinations, and Multivariate MCV Lists.

OTHER KEY STATISTICS FOR FINE-TUNING

avg_width: Average value size

byte

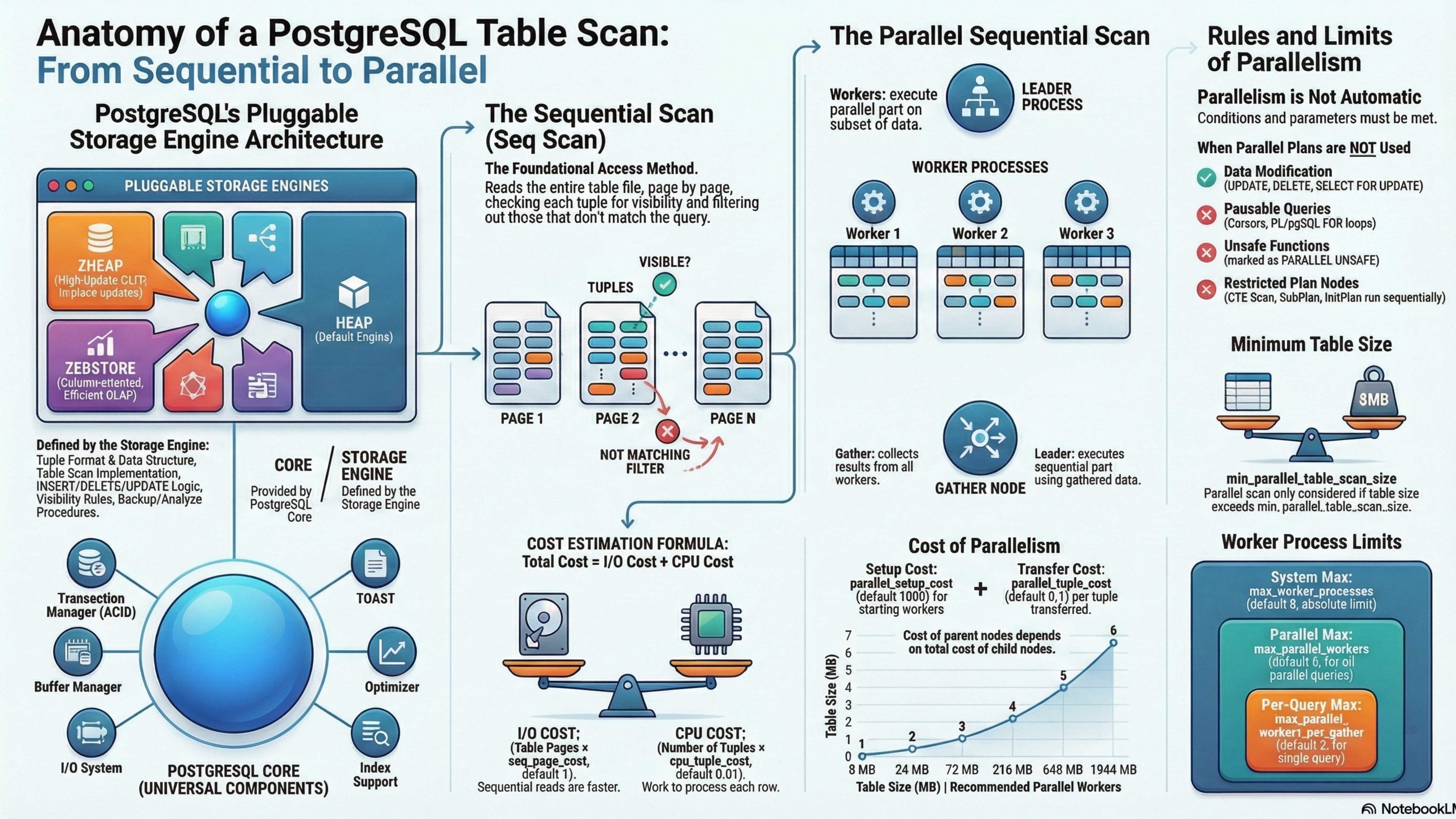
Average size of data in a column, critical for memory estimates for sorting/hashing.

correlation: Physical vs. Logical Order

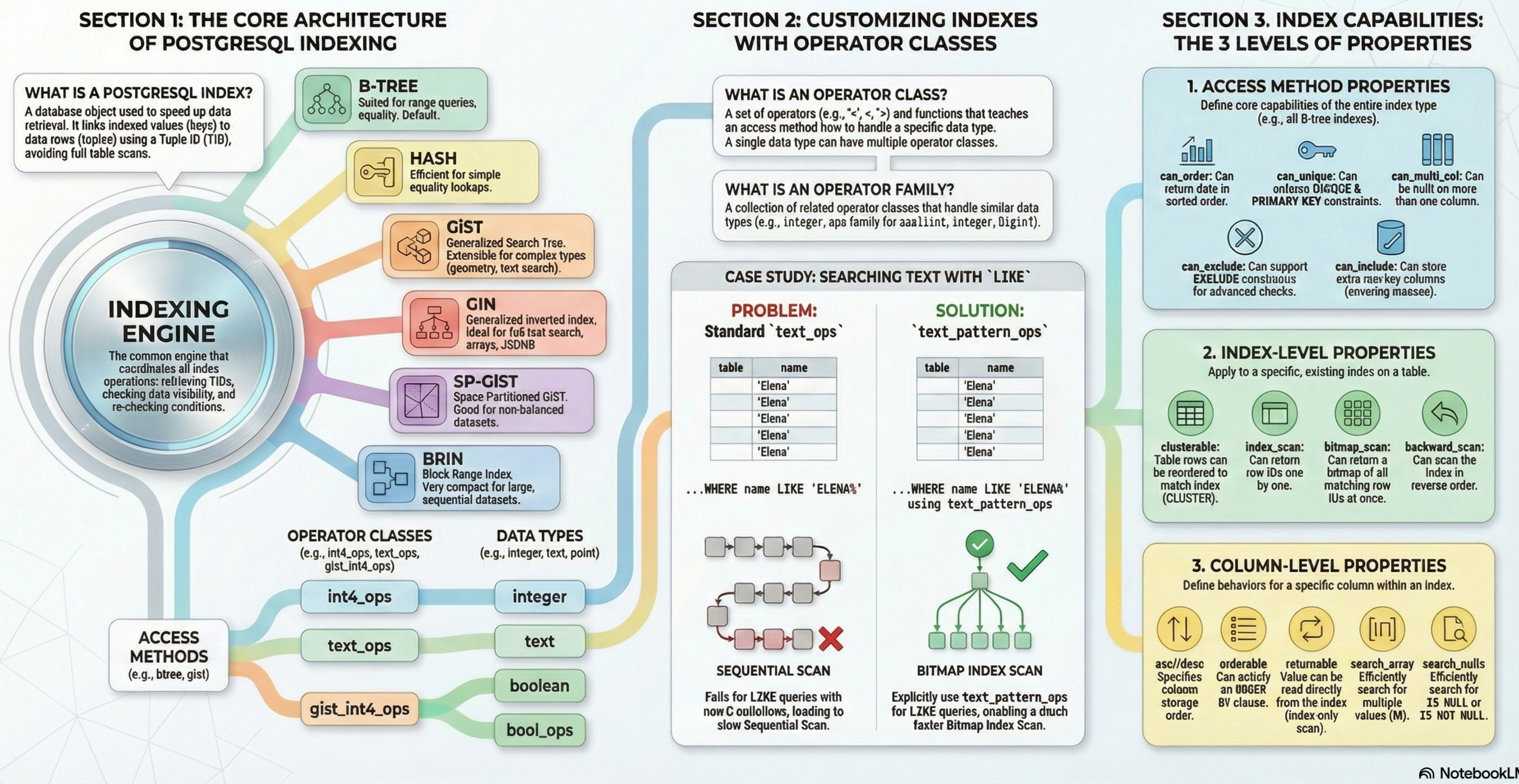
+0.9
Highly Sorted +1

Measures correlation between diss storage and logical value order. High correlation (+1 or -1) significantly speeds up index scans.

NotebookLM

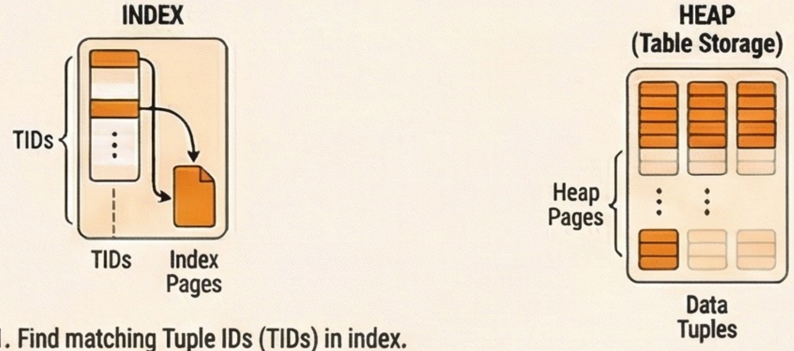

Anatomy of a PostgreSQL Table Scan:
From Sequential to Parallel
PostgreSQL's Pluggable Storage Engine Architecture
PLUGGABLE STORAGE ENGINES
ZHEAP
(High-Update CLIT; Implace updates)
HEAP
(Default Engins)
ZEBSTORE
(Culumn-ettented, Efficient OLAP)
Defined by the Storage Engine:
Tuple Format & Data Structure, Table Scan Implementation, INSERT/DELETS/UPDATE Logic, Visibility Rules, Backup/Analyze Procedures.
CORE
Provided by PostgreSQL Core
STORAGE ENGINE
Defined by the Storage Engine
Transection Manager (ACID)
Buffer Manager
I/O System
TOAST
Optimizer
Index Support
POSTGRESQL CORE (UNIVERSAL COMPONENTS)
The Sequential Scan (Seq Scan)
The Foundational Access Method.
Reads the entire table file, page by page, checking each tuple for visibility and filtering out those that don't match the query.
VISIBLE?
TUPLES
PAGE 1
PAGE 2
PAGE N
NOT MATCHING FILTER
COST ESTIMATION FORMULA:
Total Cost = I/O Cost + CPU Cost
I/O COST;
(Table Pages × seq_page_cost, default 1).
Sequential reads are faster.
CPU COST;
(Number of Tuples × cpu_tuple_cost, default 0.01).
Work to process each row.
The Parallel Sequential Scan
Workers: execute parallel part on subset of data.
LEADER PROCESS
WORKER PROCESSES
Worker 1
Worker 2
Worker 3
Gather: collects results from all workers.
GATHER NODE
Leader: executes sequential part using gathered data.
Cost of Parallelism
Setup Cost:
parallel_setup_cost (default 1000) for starting workers
+
Transfer Cost:
parallel_tuple_cost (default 0,1) per tuple transferred.
Cost of parent nodes depends on total cost of child nodes.
Table Size (MB)
0
1
2
3
4
5
6
7
8 MB
24 MB
72 MB
216 MB
648 MB
1944 MB
Table Size (MB) | Recommended Parallel Workers
Rules and Limits of Parallelism
Parallelism is Not Automatic
Conditions and parameters must be met.
When Parallel Plans are NOT Used
Data Modification
(UPDATE, DELETE, SELECT FOR UPDATE)
Pausable Queries
(Corsors, PL/pgSQL FOR loops)
Unsafe Functions
(marked as PARALLEL UNSAFE)
Restricted Plan Nodes
(CTE Scan, SubPlan, InitPlan run sequentially)
Minimum Table Size
8MB
min_parallel_table_scan_size
Parallel scan only considered if table size exceeds min. parallel..table..scan..size.
Worker Process Limits
System Max:
max_worker_processes
(default 8, absolute limit)
Parallel Max:
max_parallel_workers
(dufault 6, for oil parallel queries)
Per-Query Max:
max_parallel_worker1_per_gather
(default 2. for single query)
NotebookLM

ANATOMY OF A POSTGRESQL INDEX: THE EXTENSIBLE ENGINE



A Visual Guide to PostgreSQL Index Scans

HOW A STANDARD INDEX SCAN WORKS



1. Find matching Tuple IDs (TIDs) in index.
2. Fetch corresponding data row (tuple) from heap.

ANATOMY OF AN EXPLAIN PLAN

```
Index Scan using bookings_pkey on bookings
  Index Cond: (book_ref = '9AEBC6'::bpcher)
  Filter: (total_amount = 46500.00)
```

Checked using index — Re-checked after fetching data

THE COST OF AN INDEX SCAN

Total cost = Index page access + Heap page access & tuple processing

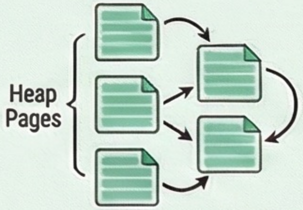
THE DECISIVE FACTOR: DATA CORRELATION

What is Correlation? Relationship between physical row order and logical index order.

HIGH CORRELATION: THE BEST CASE

(Efficient Sequential Reads)

Rows physically next to each other; efficient, sequential reads.

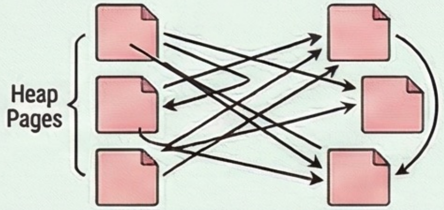


High Correlation
Near 1 or -1

LOW CORRELATION: THE WORST CASE

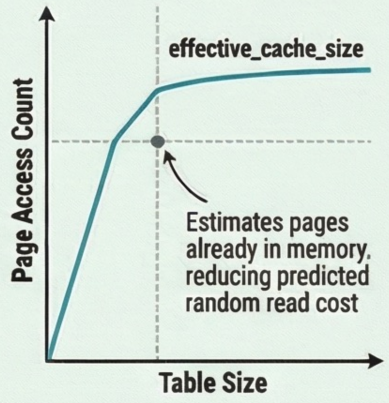
(Inefficient Random Reads)

Rows scattered randomly; many slow, random I/O operations.

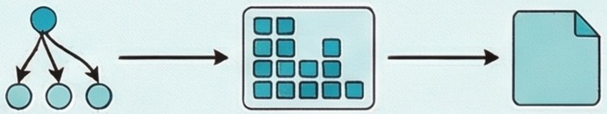


Low Correlation
Near 0

CACHING AS A PERFORMANCE BUFFER



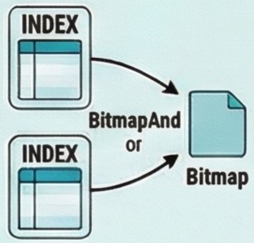
BITMAP SCANS: THE BEST OF BOTH WORLDS



A Smarter Two-Phase Approach: Collects all TIDs, sorts them by page location, then fetches pages in physical order (each page read once).

THE SOLUTION FOR LOW CORRELATION
Tums slow, randorn I/O into predictable, ordered I/O.

COMBINING MULTIPLE INDEXES

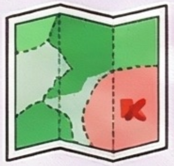


THE ROLE OF WORK_MEM

Bitmap built in memory (limited by work_mem): If space rons out, becomes 'lcesy', requiring re checks.

INDEX-ONLY SCANS: SKIPPING THE HEAP

If all required columns are in the index, heap access is avoided for significant performance boost.



THE VISIBILITY MAP CHECK
PostpreSDL checks **Visibility Map** to ensure rows are visible to current transaction.

THE IMPORTANCE OF VACUUM

Before VACUUM:
Heap Fetches 132,109
(Planner thinks heap visit needed)

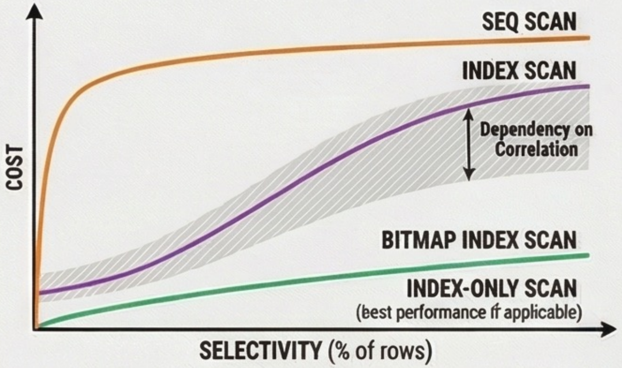
After VACUUM:
Heap Fetches 0
(Visibility Map updated, fower heap visits)

CREATING COVERING INDEXES WITH INCLUDE

Use **INCLUDE** clause to add extra columns to index without being part of the key.

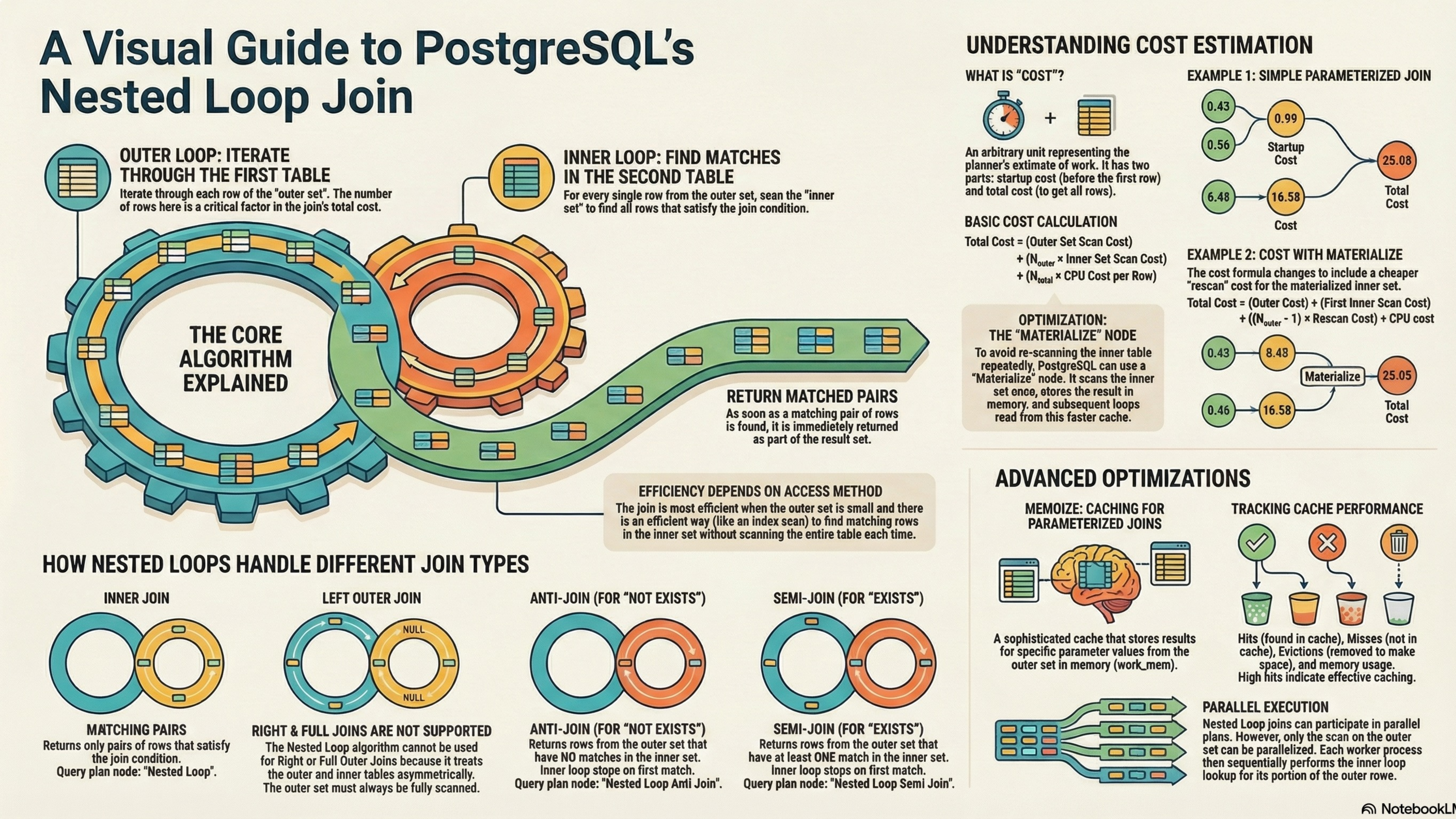
A COMPARATIVE SUMMARY

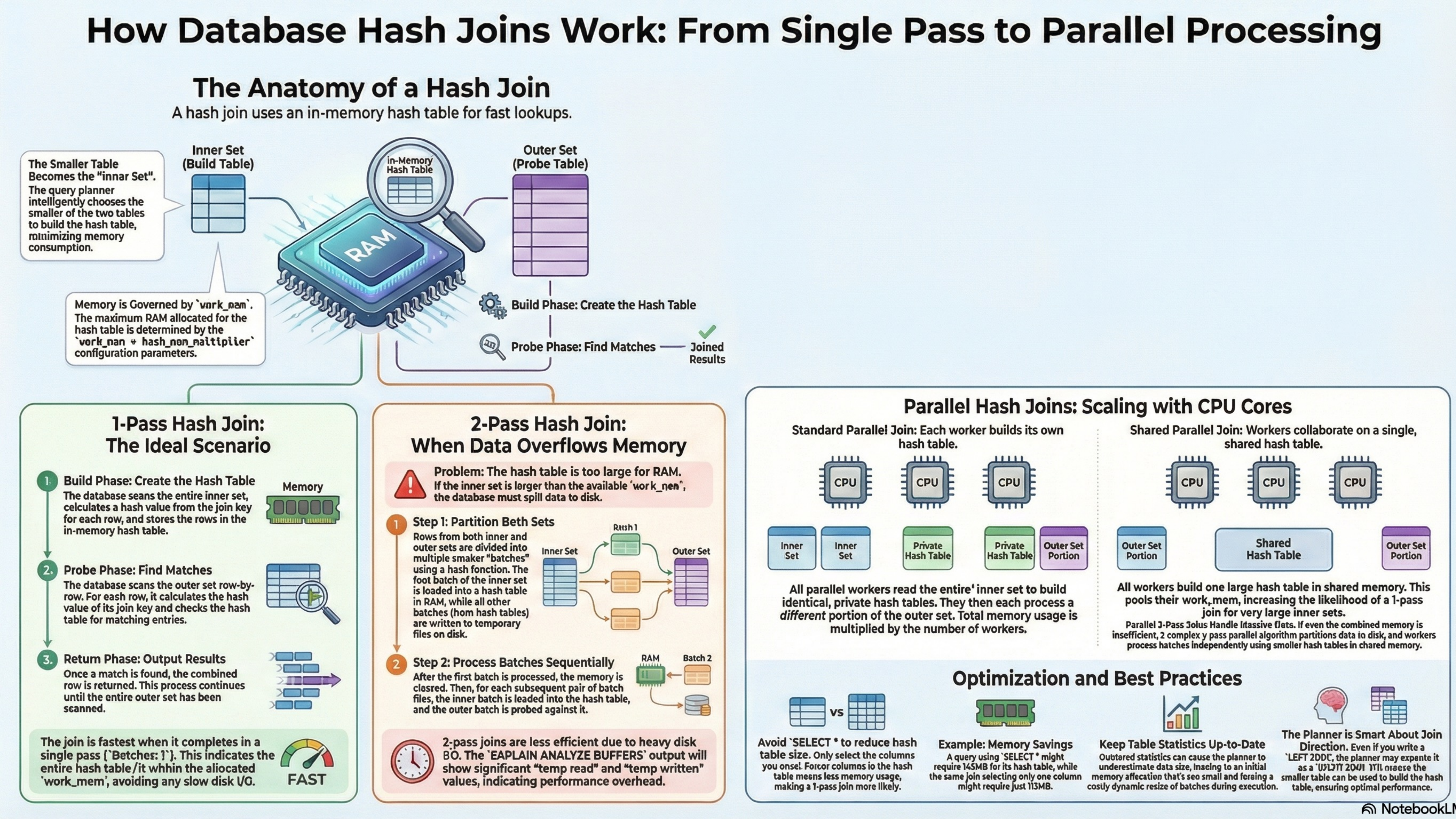
CHOOSING THE RIGHT SCAN FOR THE JOB

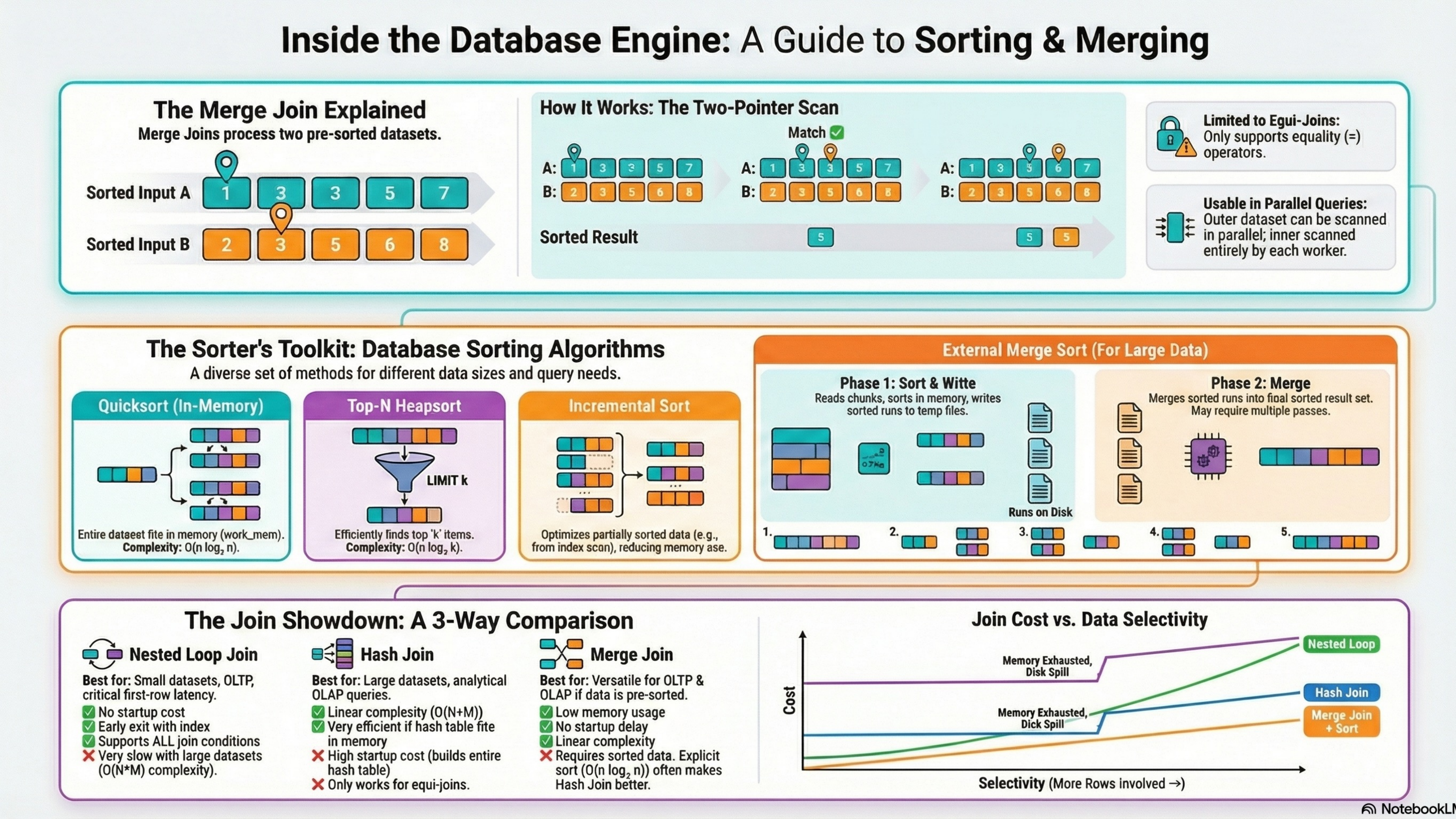


WHEN TO EXPECT EACH SCAN TYPE

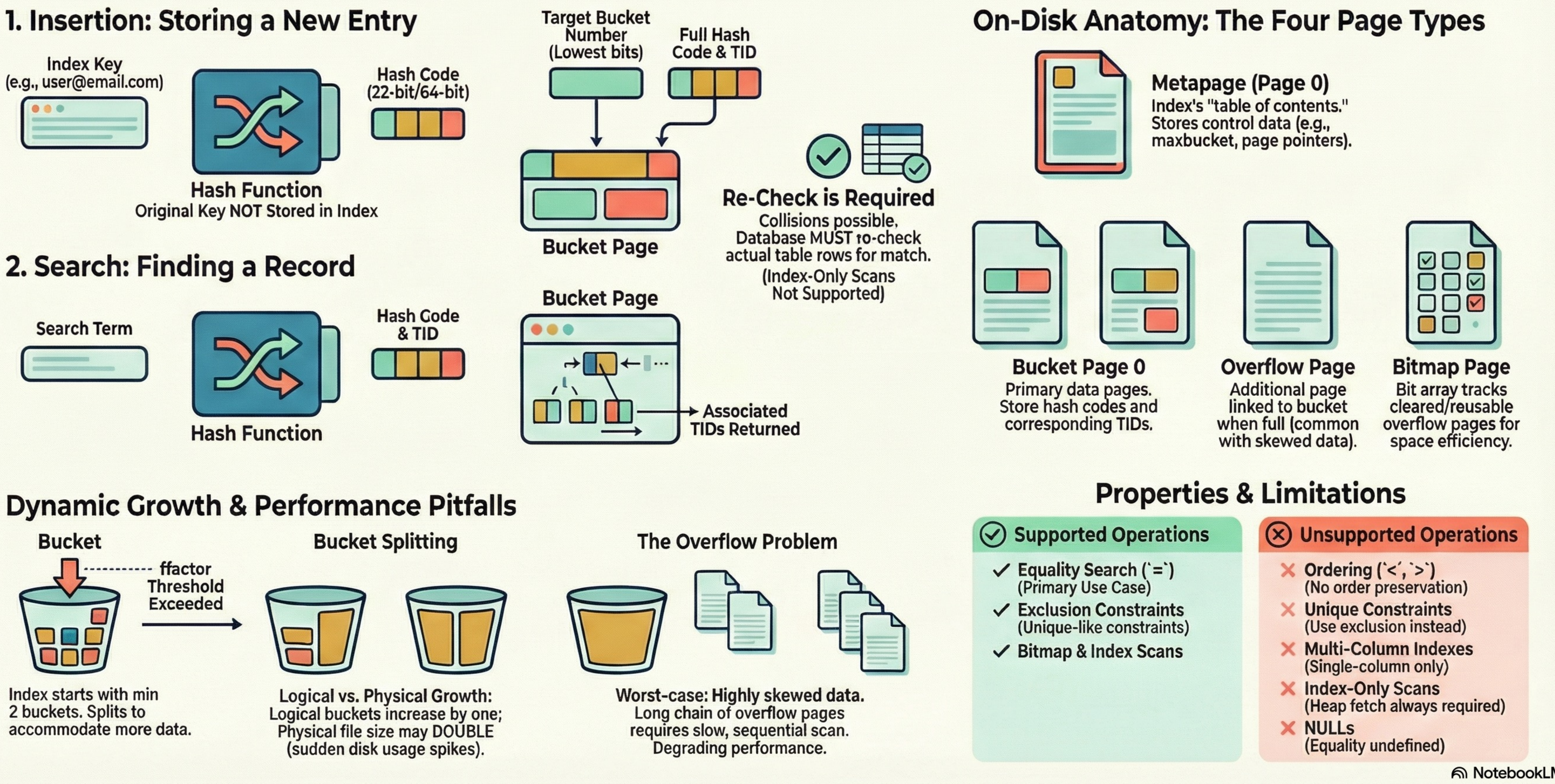
Eequential Scan	Large percentage of table read.
Index Scan	Fetching very few rows (high selectivity); degrades with low correlation.
Index-Only Scan	Extremely fast, requires covering indes and up to-date visibility map.
Bitmap Scan	Strong in low correlation; for queries too selective for seq scan but not selective enough for index scan.


A Visual Guide to PostgreSQL's Nested Loop Join
OUTER LOOP: ITERATE THROUGH THE FIRST TABLE
Iterate through each row of the "outer set". The number of rows here is a critical factor in the join's total cost.
INNER LOOP: FIND MATCHES IN THE SECOND TABLE
For every single row from the outer set, sean the "inner set" to find all rows that satisfy the join condition.
THE CORE ALGORITHM EXPLAINED
RETURN MATCHED PAIRS
As soon as a matching pair of rows is found, it is immedietely returned as part of the result set.
EFFICIENCY DEPENDS ON ACCESS METHOD
The join is most efficient when the outer set is small and there is an efficient way (like an index scan) to find matching rows in the inner set without scanning the entire table each time.
UNDERSTANDING COST ESTIMATION
WHAT IS "COST"?
+
An arbitrary unit representing the planner's extimate of work. It has two parts: startup cost (before the first row) and total cost (to get all rows).
BASIC COST CALCULATION
Total Cost = (Outer Set Scan Cost)
+ (N_{outer} × Inner Set Scan Cost)
+ (N_{total} × CPU Cost per Row)
OPTIMIZATION: THE "MATERIALIZE" NODE
To avoid re-scanning the inner table repeatedly, PostgreSQL can use a "Materialize" node. It scans the inner set ence, etores the result in memory. and subsequent loops read from this faster cache.
EXAMPLE 1: SIMPLE PARAMETERIZED JOIN
0.43
0.56
0.99
Startup Cost
6.48
16.58
Cost
25.08
Total Cost
EXAMPLE 2: COST WITH MATERIALIZE
The cost formula changes to include a cheaper "rescan" cost for the materialized inner set.
Total Cost = (Outer Cost) + (First Inner Scan Cost)
+ ((N_{outer} - 1) × Rescan Cost) + CPU cost
0.43
8.48
Materialize
25.05
0.46
16.58
Total Cost
HOW NESTED LOOPS HANDLE DIFFERENT JOIN TYPES
INNER JOIN
MATCHING PAIRS
Returns only pairs of rows that satisfy the join condition.
Query plan node: "Nested Loop".
LEFT OUTER JOIN
NULL
NULL
RIGHT & FULL JOINS ARE NOT SUPPORTED
The Nested Loop algorithm cannot be used for Right or Full Outer Joins because it treats the outer and inner tables asymmetrically. The outer set must always be fully scanned.
ANTI-JOIN (FOR "NOT EXISTS")
ANTI-JOIN (FOR "NOT EXISTS")
Returns rows from the outer set that have NO matches in the inner set. Inner loop stope on first match.
Query plan node: "Nested Loop Anti Join".
SEMI-JOIN (FOR "EXISTS")
SEMI-JOIN (FOR "EXISTS")
Returns rows from the outer set that have at least ONE match in the inner set. Inner loop stops on first match.
Query plan node: "Nested Loop Semi Join".
ADVANCED OPTIMIZATIONS
MEMOIZE: CACHING FOR PARAMETERIZED JOINS
A sophisticated cache that stores results for specific parameter values from the outer set in memory (work_mem).
TRACKING CACHE PERFORMANCE
Hits (found in cache), Misses (not in cache), Evictions (removed to make space), and memory usage. High hits indicate effective caching.
PARALLEL EXECUTION
Nested Loop joins can participate in parallel plans. However, only the scan on the outer set can be parallelized. Each worker process then sequentially performs the inner loop lookup for its portion of the outer rowe.
NotebookLM


How Database Hash Joins Work: From Single Pass to Parallel Processing
The Anatomy of a Hash Join
A hash join uses an in-memory hash table for fast lookups.
The Smaller Table Becomes the "innar Set". The query planner intelligently chooses the smaller of the two tables to build the hash table, minimizing memory consumption.
Inner Set (Build Table)
in-Memory Hash Table
RAM
Outer Set (Probe Table)
Memory is Governed by `vork_mam`. The maximum RAM allocated for the hash table is determined by the `vork_man + hash_nem_maltipiier` configuration parameters.
Build Phase: Create the Hash Table
Probe Phase: Find Matches
Joined Results
1-Pass Hash Join: The Ideal Scenario
1. Build Phase: Create the Hash Table
The database seans the entire inner set, calculates a hash value from the join key for each row, and stores the rows in the in-memory hash table.
Memory
2. Probe Phase: Find Matches
The database scans the outer set row-by-row. For eash row, it calculates the hash value of its join key and checks the hash table for matching entries.
3. Retum Phase: Output Results
Once a match is found, the combined row is returned. This process continues until the entire outer set has been seanned.
The join is fastest when it completes in a single pass (`Betches: 1`). This indicates the entire hash table/it whhin the aliocated `work_mem`, avoiding any slow disk I/G.
FAST
2-Pass Hash Join: When Data Overflows Memory
Problem: The hash table is too large for RAM.
If the inner set is larger than the available `work_nen`, the database must spill data to disk.
1 Step 1: Partition Beth Sets
Rows from both inner and outer sets are divided into multiple smaker "batches" using a hash fonction. The foot batch of the inner set is loaded into a hash table in RAM, while all other batches (hom hash tables) are written to temporary files on disk.
Inner Set
Ratsh 1
Outer Set
2 Step 2: Process Batches Sequentially
After the first batch is processed, the memory is clesred. Then, for each subsequent pair of batch files, the inner batch is leaded into the hash table, and the outer batch is probed against it.
RAM
Batch 2
2-pass joins are less efficient due to heavy disk BO. The `EAPLAIN ANALYZE BUFFERS` output will show significant "temp read" and "temp written" values, indicating performance overhead.
Parallel Hash Joins: Scaling with CPU Cores
Standard Parallel Join: Each worker builds its own hash table.
CPU
CPU
CPU
Inner Set
Inner Set
Private Hash Table
Private Hash Table
Outer Set Portion
All parallel workers read the entire inner set to build identical, private hash tables. They then each process a different portion of the outer set. Total memory usage is multiplied by the number of workers.
Shared Parallel Join: Workers collaborate on a single, shared hash table.
CPU
CPU
CPU
Outer Set Portion
Shared Hash Table
Outer Set Portion
All workers build one large hash table in shared memory. This pools their work_mem, increasing the likelihood of a 1-pass join for very large inner sets.
Parallel 2-Pass Jolus Handle Massive Clats. If even the combined memory is insefficient, 2 complex y pass parallel algorithm partitions data in disk, and workers process hatches independently using smoller hash tables in chared memory.
Optimization and Best Practices
vs
Avoid `SELECT *` to reduce hash table size. Only select the columns you onsel. Forcor columns io the hash table meens less memory usage, making a 1-pass join more likely.
Example: Memory Savings
A query using `SELECT *` might require 145MB for its hash table, while the same join selecting only one column might require just 113MB.
Keep Table Statistics Up-to-Date
Outdated statistics can cause the planner to underestimate data size, lnaeing to an initial memory affecation that's seo small and foreing a costly dynamic resize of batches during execution.
The Planner is Smart About Join Direction.
Even if you write a `LEFT 2DDL`, the planner may expente it as a `ULJIT 2OUI` ITIL anaese the smaller table can be used to build the hash table, ensuring optimal performance.
NotebookLM

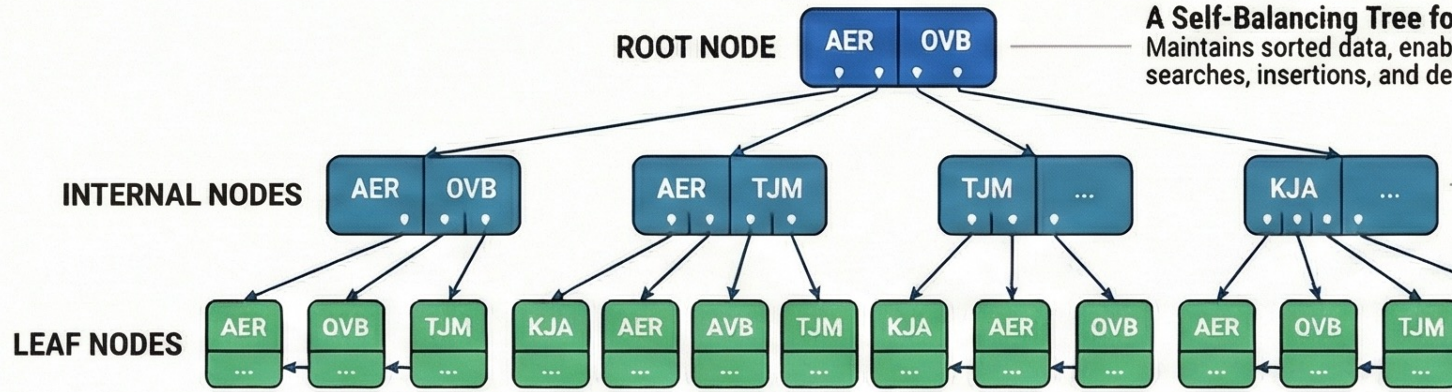

Inside the Database Engine: A Guide to Sorting & Merging
The Merge Join Explained
Merge Joins process two pre-sorted datasets.
Sorted Input A
1 3 3 5 7
Sorted Input B
2 3 5 6 8
How It Works: The Two-Pointer Scan
Match
A: 1 3 3 5 7
B: 2 3 5 6 8
Sorted Result
5
Limited to Equi-Joins: Only supports equality (=) operators.
Usable in Parallel Queries: Outer dataset can be scanned in parallel; inner scanned entirely by each worker.
The Sorter's Toolkit: Database Sorting Algorithms
A diverse set of methods for different data sizes and query needs.
Quicksort (In-Memory)
Entire dataset fits in memory (work_mem).
Complexity: O(n log2 n).
Top-N Heapsort
LIMIT k
Efficiently finds top 'k' items.
Complexity: O(n log2 k).
Incremental Sort
Optimizes partially sorted data (e.g., from index scan), reducing memory use.
External Merge Sort (For Large Data)
Phase 1: Sort & Witte
Reads chunks, sorts in memory, writes sorted runs to temp files.
Runs on Disk
Phase 2: Merge
Merges sorted runs into final sorted result set. May require multiple passes.
1.
2.
3.
4.
5.
The Join Showdown: A 3-Way Comparison
Nested Loop Join
Best for: Small datasets, OLTP, critical first-row latency.
No startup cost
Early exit with index
Supports ALL join conditions
Very slow with large datasets (O(N*M) complexity).
Hash Join
Best for: Large datasets, analytical OLAP queries.
Linear complesity (O(N+M))
Very efficient if hash table fite in memory
High startup cost (builds entire hash table)
Only works for equi-joins.
Merge Join
Best for: Versatile for OLTP & OLAP if data is pre-sorted.
Low memory usage
No startup delay
Linear complexity
Requires sorted data. Explicit sort (O(n log2 n)) often makes Hash Join better.
Join Cost vs. Data Selectivity
Memory Exhausted, Disk Spill
Memory Exhausted, Dick Spill
Cost
Selectivity (More Rows involved →)
Nested Loop
Hash Join
Merge Join + Sort
NotebookLM

A Deep Dive into PostgreSQL Hash Indexes



A Visual Guide to B-Trees in PostgreSQL

ANATOMY OF A B-TREE



A Self-Balancing Tree for Sorted Data: Maintains sorted data, enabling efficient searches, insertions, and deletions.

Short and "Bushy": Each node has many children (hundreds), resulting in a shallow tree, minimizing disk reads.

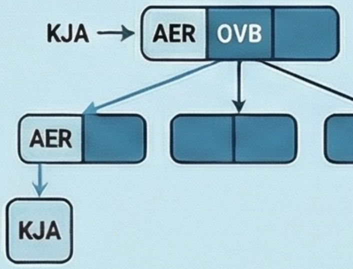
Always Sorted: Data is sorted within each node and across levels. Leaf nodes are a doubly-linked list for fast ordered scans.

Perfectly Balanced: All leaf nodes at same depth for consistent search time.

HOW B-TREES HANDLE DATA

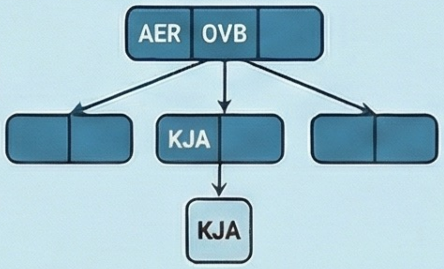
1. Searching: Start at the Root

Compare search value to keye to decide child branch (AER ≤ KJA < OVB).



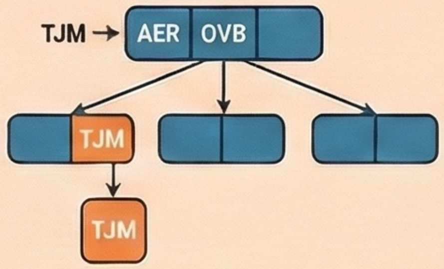
2. Searching: Descend the Tree

Repeatedly narrow search path until a leaf node is reached, fast due to shallow depth.



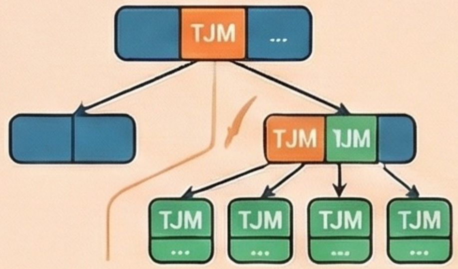
3. Insertion: Find the Spot

Traverse tree to find correct leaf node to maintain sorted order.



4. Insertion: Split if Full

If target leaf is full, split it and add a reference to the parent. Split can propagate to the root, growing the tree.



B-TREE PROPERTIES

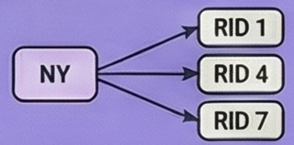
Supports Ordering & Uniqueness
Only access method in PostgreSQL to enforce data aniqueness and retum sorted data.

Full Scan Capabilities
Supports Index Scans, Bitmap Scans, and Backward Scans (for DESC queries) via leal node linked list.

Search Flexibility
Supports searching for NULL values and retrieval directly from index (index-only scans).

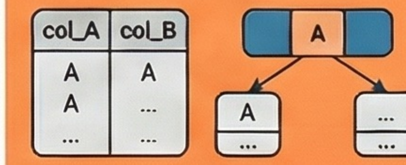
ADVANCED FEATURES IN POSTGRESQL

Deduplication for Efficiency



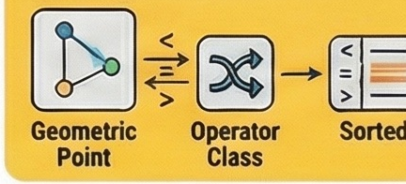
Stores single key with a list of pointers for non-unique indexes, reducing index size.

Suffix Truncation



Internal nodes may omit trailing columns in multi:column indexes to save page space.

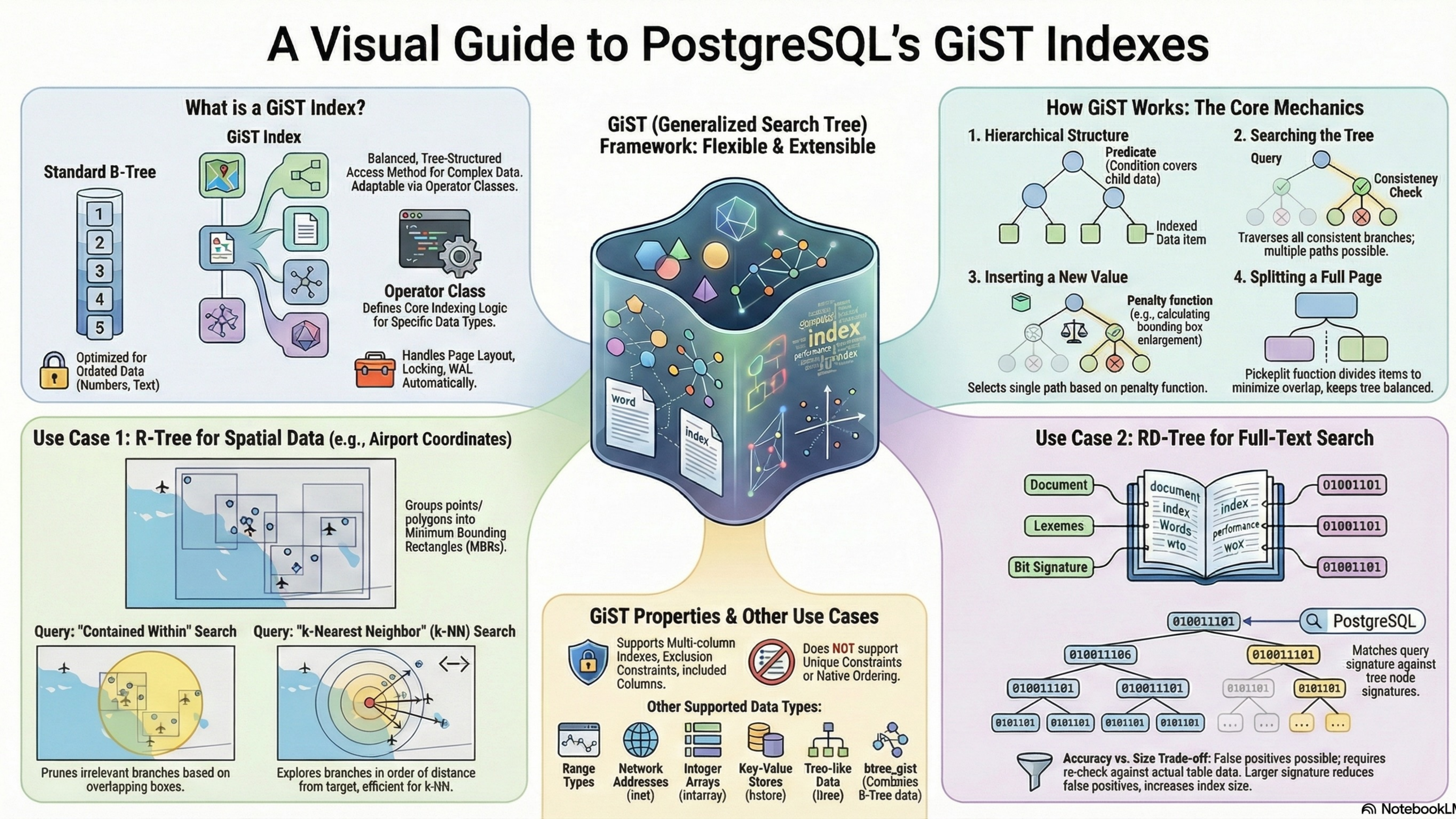
Custom Sorting with Operator Classes



Multi-Column Indexing Order Matters

An index on (col_A, col_B) can be used efficiently for searches on col_A or on (col_A and col_B), but not for searches on col_B alone. Column order is critical for performance.

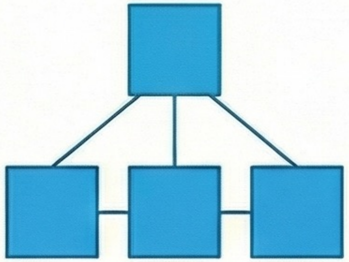
NotebookLM

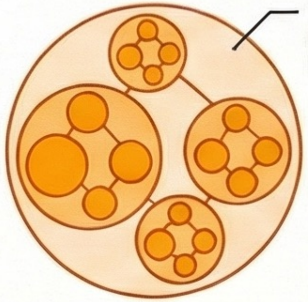

A Visual Guide to PostgreSQL's GiST Indexes
What is a GiST Index?
GiST Index
Standard B-Tree
1
2
3
4
5
Optimized for Ordated Data (Numbers, Text)
Balanced, Tree-Structured Access Method for Complex Data. Adaptable via Operator Classes.
Operator Class
Defines Core Indexing Logic for Specific Data Types.
Handles Page Layout, Locking, WAL Automatically.
GiST (Generalized Search Tree) Framework: Flexible & Extensible
Word
index
performance
How GiST Works: The Core Mechanics
1. Hierarchical Structure
Predicate (Condition covers child data)
Indexed Data item
2. Searching the Tree
Query
Consisteney Check
Traverses all consistent branches; multiple paths possible.
3. Inserting a New Value
Penalty function (e.g., calculating bounding box enlargement)
Selects single path based on penalty function.
4. Splitting a Full Page
Pickeplit function divides items to minimize overlap, keeps tree balanced.
Use Case 1: R-Tree for Spatial Data (e.g., Airport Coordinates)
Groups points/ polygons into Minimum Bounding Rectangles (MBRs).
Query: "Contained Within" Search
Prunes irrelevant branches based on overlapping boxes.
Query: "k-Nearest Neighbor" (k-NN) Search
Explores branches in order of distance from target, efficient for k-NN.
Use Case 2: RD-Tree for Full-Text Search
Document
Lexemes
Bit Signature
document
index
Words
wto
performance
wox
01001101
01001101
01001101
PostgreSQL
010011101
010011106
010011101
010011101
010011101
0101101
0101101
0101101
0101101
0101101
0101101
0101101
...
...
...
...
Matches query signature against tree node signatures.
Accuracy vs. Size Trade-off: False positives possible; requires re-check against actual table data. Larger signature reduces false positives, increases index size.
GiST Properties & Other Use Cases
Supports Multi-column Indexes, Exclusion Constraints, included Columns.
Does NOT support Unique Constraints or Native Ordering.
Other Supported Data Types:
Range Types
Network Addresses (inet)
Intoger Arrays (intarray)
Key-Value Stores (hstore)
Treo-like Data (lree)
btree_gist (Combines B-Tree data)
NotebookLM

SP-GiST Explained: A Visual Guide to Space-Partitioned Indexing

A framework for creating unbalanced, space-partitioned trees optimized for specialized data types like spatial points and text strings.

What is an SP-GiST Index?



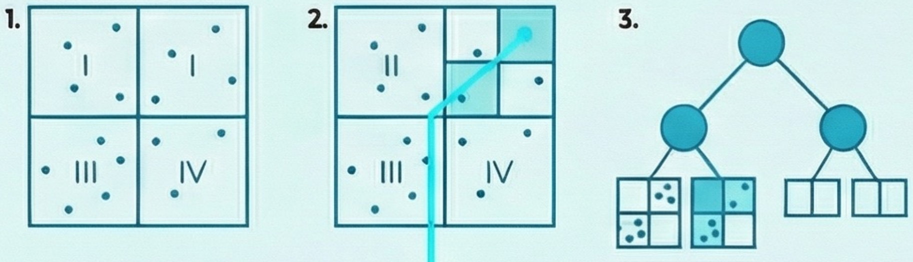


Internal nodes contain a shared condition (prefix); Leaf nodesets & TIDs.

❌ **B-tree/GiST Nodes:** One Node per Page, Balanced

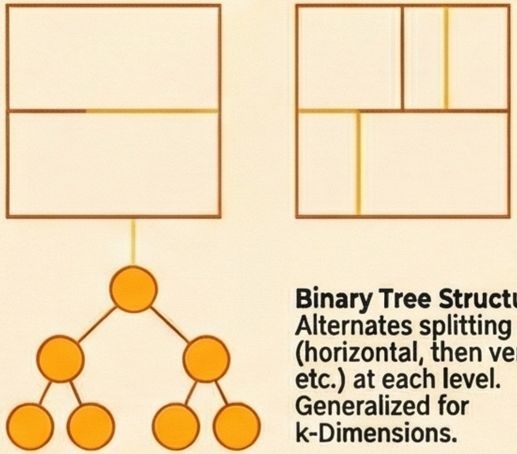
✅ **SP-GiST:** Packed, Small Nodes on Single Page, Unbalanced Structure (Branch depths vary)

Use Case 1: Quad-tree for 2D Points



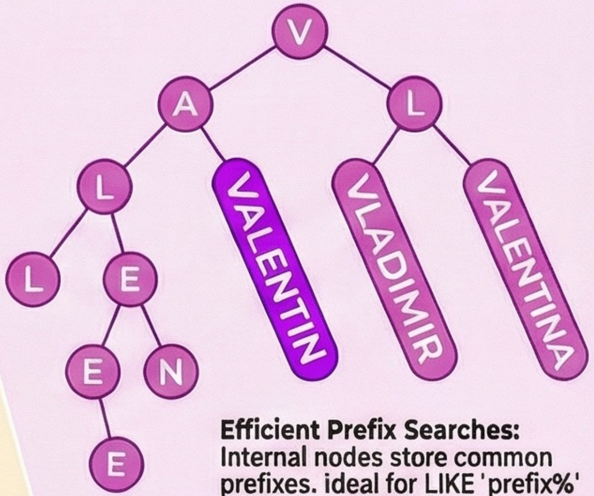
Searching: Prunes irrelevant quadrants using a "consistency function" to find points above (3,7). New points added to corresponding quadrants, triggering "picksplit" when full.

Use Case 2: k-D Tree for Points



Binary Tree Structure: Alternates splitting axis (horizontal, then vertical, etc.) at each level. Generalized for k-Dimensions.

Use Case 3: Radix Tree for Strings



Efficient Prefix Searches: Internal nodes store common prefixes. ideal for LIKE 'prefix%' or starts_with(). More compact than B-frees, only stores necessary parts of strings.

Key SP-GiST Properties: Supported vs. Not Supported

Feature	Supported by SP-GiST	Description
Exclusion Constraints	Yes (✅)	Can enforce constraints like "no overlapping ranges".
Covering Indexes (INCLUDE)	Yes (✅)	Can include non-key columns for index-only scans.
Bitmap Scans	Yes (✅)	Efficiently combines results of multiple conditions.
NULL Value Indexing	Yes (✅)	NULLs are supported in a separate tree structure.
Ordering Results	No (❌)	Cannot return results in sorted order directly from index.
Unique Constraints	No (❌)	Cannot be used to enforce uniqueness.
Multi-Column Indexes	No (❌)	An SP-GiST index can only be created on a single column.
Clustering	No (❌)	Cannot be used for the CLUSTER command.

Unlocking PostgreSQL: A Deep Dive into GIN Indexes

What is a GIN Index?

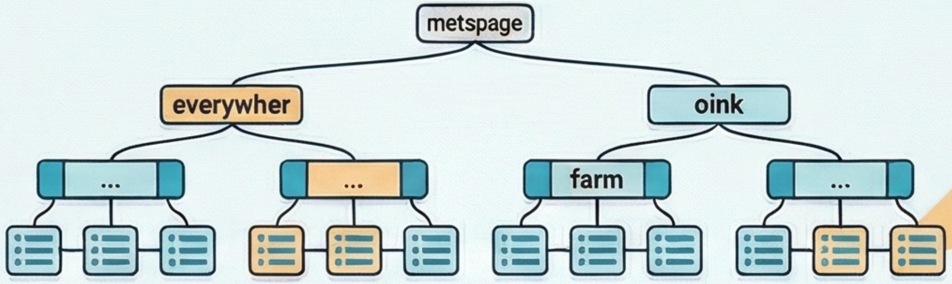
GIN stands for Generalized Inverted Index.

It's designed for composite data types composed of separate elements, like words in a document or items in an array.

It indexes the elements, not the whole value.

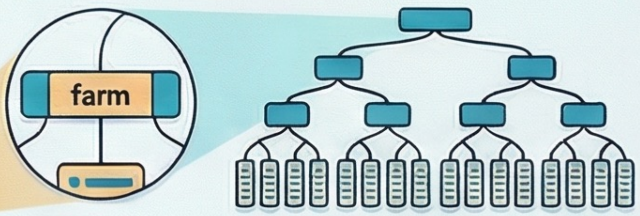
GIN maps each individual element to all the table rows (TIDs) that contain it, similar to a book's index.

The Core Structure is a B-Tree of Elements.



The main data structure is a B-tree where keys are the indexed elements (e.g., words). Each leaf entry in this tree points to a list of TIDs where that element appears.

Lists of row IDs are called "Posting Lists".



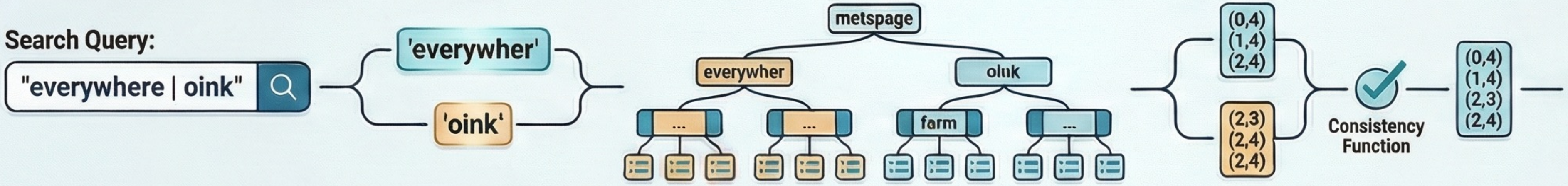
If a posting list for an element becomes too long (i.e., the element is very common), it's stored in a separate B-tree called a "posting tree" for efficiency.

GIN in Action: A Full-Text Search Example

1 Extract Keys from the Query.

2 Find TIDs for Each Key.

3 Merge and Check Results.

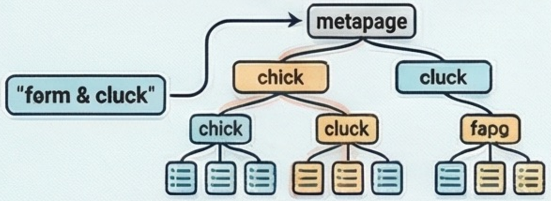


Consistency Check Process for Query: "everywhere | oink"

TID	Contains "everywhere"	Contains "oink"	Consistency Function (result)
(0,4)	✓	—	✓
(1,4)	✓	—	✓
(2,3)	—	✓	✓
(2,4)	✓	✓	✓

Performance & Trade-offs

Search is optimized by prioritizing rare terms.



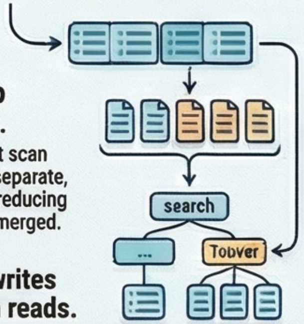
For a query like 'farm & cluck', GIN fxet finds the few documents with the rare word ('cluck') and only then checks if those specific documents also contain the common word ('farm'), saving significant work.

Querying for a rare term is drastically faster.

Searching for 'wrote' (231,173 docs) took 243ms, while searching for 'wrote & tattoe' (1 doc) took only 8 6ms, almost as fast as searching for 'tattoo' alone (2.2me).

GIN updates can be slow.

Indexing a single document can require many changes across the index tree, as each word in the document is a separate entry.

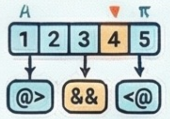


"Fast Update" speeds up writes by delaying them.

During a search, PostgreSQL must scan both the main index tree and the separate, unsorted pending list, potentially reducing read performance until the list is merged.

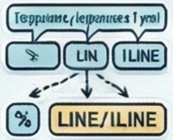
The pending list makes writes faster but can slow down reads.

Key Use Cases



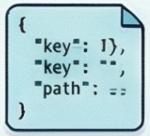
Indexing Arrays

Speeds up queries that check for element containment, overlap, or if an array is contained by another.



Fuzzy String Search with Trigrams

The pg_figm extension allows GIN to index three character segments of test, enabling very test similarity searches and pattern matching.



Indexing JSONB Documents

GIN offers two operator classes for JSONB: jaoab_aps (default) indexes every key and value, while jsook_path_ape indexes the full path to each value, which is often more efficient.

GIN Index Properties & Limitations

Strengths

- Supports multi-column indexes.
- Returns results via BITMAP SCAN.
- Perfect for checking the esistance of elements within composite types.

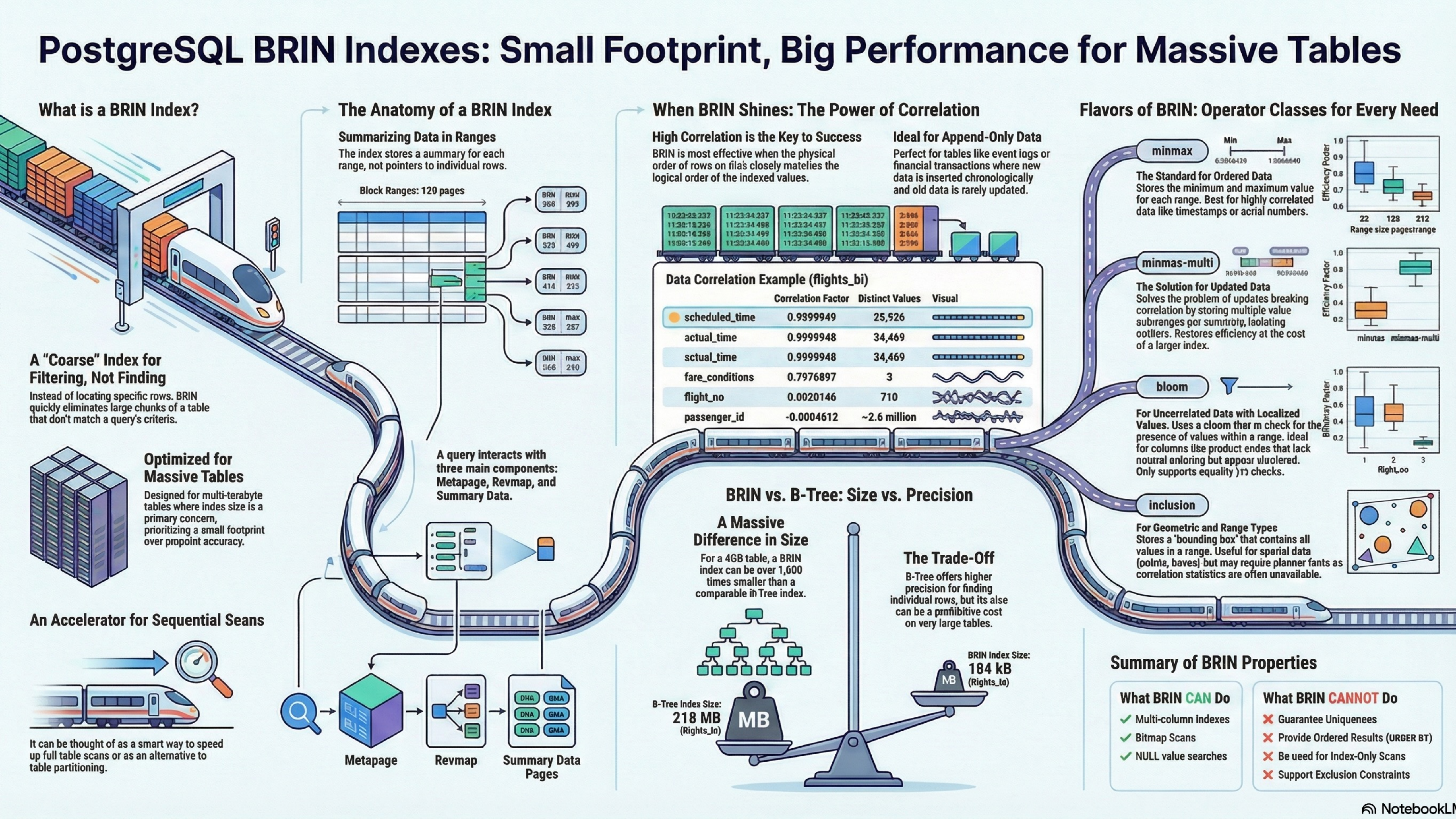
Weaknesses

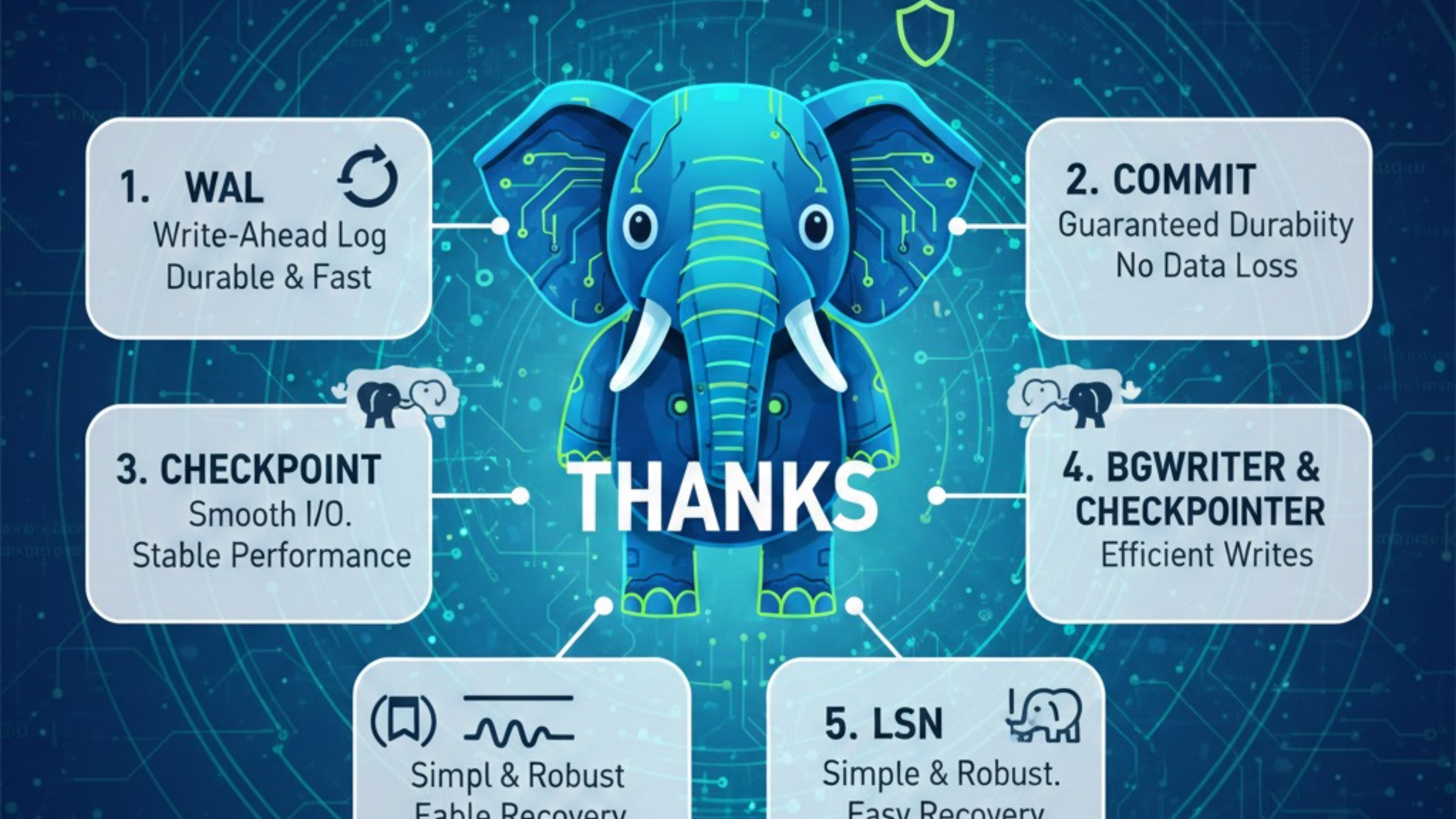
- Cannot enforce UNIQUE constraints.
- Does not support ordering, so it can't be nsed to avoid sorting steps.
- Index only scans are not possible.

Inefficient with LIMIT clauses.

GIN always builds a full bitmap of all matching rows before fetching any, making it inefficient to tolok just foe 'bral W' results.

Alternative: RUM Index. The RUM extension is based on GIN but adds features GIN lacks, like storing positional information (for phrase searching) and supporting ordering, at the cost of larger index size ar


PostgreSQL BRIN Indexes: Small Footprint, Big Performance for Massive Tables
What is a BRIN Index?
A "Coarse" Index for Filtering, Not Finding
Instead of locating specific rows. BRIN quickly eliminates large chunks of a table that don't match a query's criteris.
Optimized for Massive Tables
Designed for multi-terabyte tables where indes size is a primary concern, prioritizing a small footprint over propoint accuracy.
An Accelerator for Sequential Seans
It can be thought of as a smart way to speed up full table scans or as an alternative to table partitioning.
The Anatomy of a BRIN Index
Summarizing Data in Ranges
The index stores a aummary for each range, not pointers to individual rows.
Block Ranges: 120 pages
A query interacts with three main components: Metapage, Revmap, and Summary Data.
Metapage
Revmap
Summary Data Pages
When BRIN Shines: The Power of Correlation
High Correlation is the Key to Success
BRIN is most effective when the physical order of rows on fila's closely mateiies the logical order of the indexed values.
Ideal for Append-Only Data
Perfect for tables like event iogs or financial transactions where new data is inserted chronologically and old data is rarely updated.
Data Correlation Example (flights_bi)
Correlation Factor | Distinct Values | Visual
scheduled_time | 0.9399949 | 25,926
actual_time | 0.9999948 | 34,469
sctual_time | 0.9999948 | 34,469
fare_conditions | 0.7976897 | 3
flight_no | 0.0020146 | 710
passenger_id | -0.0004612 | ~2.6 million
BRIN vs. B-Tree: Size vs. Precision
A Massive Difference in Size
For a 4GB table, a BRIN index can be over 1,600 times smaller than a comparable in Tree index.
The Trade-Off
B-Tree offers higher precision for finding individual rows, but its alse can be a prnfiibitive cost on very large tables.
BRIN Index Size:
184 kB
(Rights_to)
B-Tree Indes Size:
218 MB
(Rights_la)
MB
Flavors of BRIN: Operator Classes for Every Need
minmax
The Standard for Ordered Data
Stores the minimum and maximum value for each range. Best for highly correlated data like timestamps or acrial numbers.
Range size pagestrange
minmas-multi
The Solution for Updated Data
Solves the problem of updates breaking correlation by storing multiple value subiranges por sumrtoty, laolating ootliers. Restores efficiency at the cost of a larger index.
minmas
minmas-multi
bloom
For Uncerrelated Data with Localized Values. Uses a cloom ther m check for the presence of values within a range. ideal for columns use product endes that lack noutral enloring but appor viuolered. Only supports equality)r3 checks.
inclusion
For Geometric and Range Types
Stores a 'bounding box' that contains all values in a range. Useful for sperial data (polnis, baves) but may require planner fants as correlation statistics are often unavailable.
Summary of BRIN Properties
What BRIN CAN Do
✔ Multi-column Indexes
✔ Bitmap Scans
✔ NULL value searches
What BRIN CANNOT Do
✘ Guarantee Uniquenees
✘ Provide Ordered Results (URGER BT)
✘ Be ueed for Index-Only Scans
✘ Support Exclusion Constraints
NotebookLM


1. WAL
Write-Ahead Log
Durable & Fast
2. COMMIT
Guaranteed Durabiity
No Data Loss
3. CHECKPOINT
Smooth I/O.
Stable Performance
THANKS
4. BGWRITER &
CHECKPOINTER
Efficient Writes
Simpl & Robust
5. LSN
Simple & Robust.