Visualizing PostgreSQL Storage Internals

PG DATA 2026

boringSQL

boringSQL

radim@boringsql.com

Skeptic, Turned Believer

PG DATA 2026

boringSQL

PG Data 2026 Sponsors

Platinum Sponsors

aws

redgate

Gold Sponsors

Microsoft

FUJITSU

pganalyze

EDB POSTGRES AI

Google Cloud

Bronze Sponsors

pgEdge

COMMAND PROMPT, INC.

STORMATICS
Excellence in PostgreSQL Services

caktus GROUP

Community Partners

Open Alliance For PostgreSQL Education

Chicago aws user group

CODE & COFFEE

DATA IN THE D
detroit's data community

WOMEN IN DATA™

PgUs

Why should you care?

This is not academic knowledge

Table bloat

VACUUM

Performance

Concurrency

DELETEs and UPDATEs

boringSQL

The toolkit

The pg-storage-visualizer

github.com/boringSQL/pg-storage-visualizer

pageinspect
pgstattuple

PG DATA 2026

boringSQL

Act I

Why does a simple UPDATE create a whole new row?

PG DATA 2026

boringSQL

Tuple vs Row

Row is a logical concept (SQL)

Tuple is the physical concept

1:N (over time)

PostgreSQL does not care about your rows

It cares about 8KB blocks.

PG DATA 2026

boringSQL

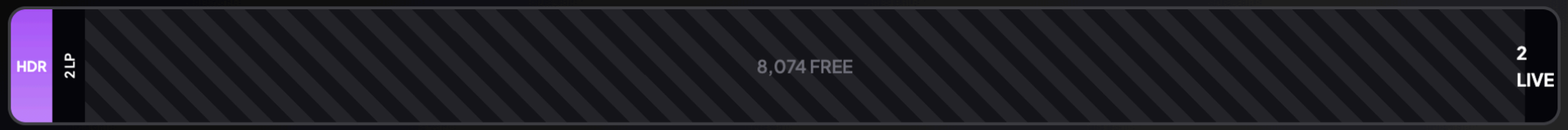
```
SHOW block_size;

block_size
------------
 8192
```

Heap Page 0 Table: demo

Page 1 → ×

8KB Heap Page (8,192 bytes)

1% Full



Live Tuples	Dead Tuples	Free Space	Line Pointers
2	0	7.9 KB	2

Heap Tuples

● 2 live

lp=1 LP_NORMAL (0,1)

xmin (insert)	xmax (delete)
406999	–

ctid (next version) (0,1)

Size 43 bytes

COMMITTED

lp=2 LP_NORMAL (0,2)

xmin (insert)	xmax (delete)
406999	–

ctid (next version) (0,2)

Size 43 bytes

COMMITTED

Page Header Line Pointers Data

PG DATA 2026

boringSQL

Every row carries 23 bytes of overhead

HeapTupleHeaderData struct

Field	Size	What it does
t_xmin	4B	Transaction that **created** this row
t_xmax	4B	Transaction that **deleted/updated** this row
t_ctid	6B	Physical location (self-pointer, or forward link)
t_infomask	2B	16 hint bits (XMIN_COMMITTED, HAS_NULL, ...)
t_infomask2	2B	Number of columns + HOT flags
t_hoff	1B	Offset to actual data
null bitmap	var	1 bit per column (if any NULLs)

t_xmin + t_xmax = the MVCC fingerprint

Eight bytes to rule them all, and in PostgreSQL bind them

PG DATA 2026

boringSQL

An UPDATE is a DELETE + INSERT

```
Before UPDATE:
  Row v1: xmin=100, xmax=0, ctid=(0,1)    <- visible

After UPDATE:
  Row v1: xmin=100, xmax=200, ctid=(0,2)  <- dead, "deleted" by tx 200
  Row v2: xmin=200, xmax=0,   ctid=(0,2)  <- new version, created by tx 200
```

Act II

Why does a 2GB table use 20GB on disk?

PG DATA 2026

boringSQL

Dead tuples don't clean themselves up

One UPDATE = one corpse.
A million UPDATEs = ?

PG DATA 2026

boringSQL

Enter VACUUM

The cleanup crew

PG DATA 2026

boringSQL

Why your 2GB table is 20GB

Reusable ≠ returned

Dead space is marked reusable (Free Space Map)

OS only gets space back from **empty pages at the tail**

REPACK is the only option

boringSQL

Act III

Why does VACUUM seem to do nothing?

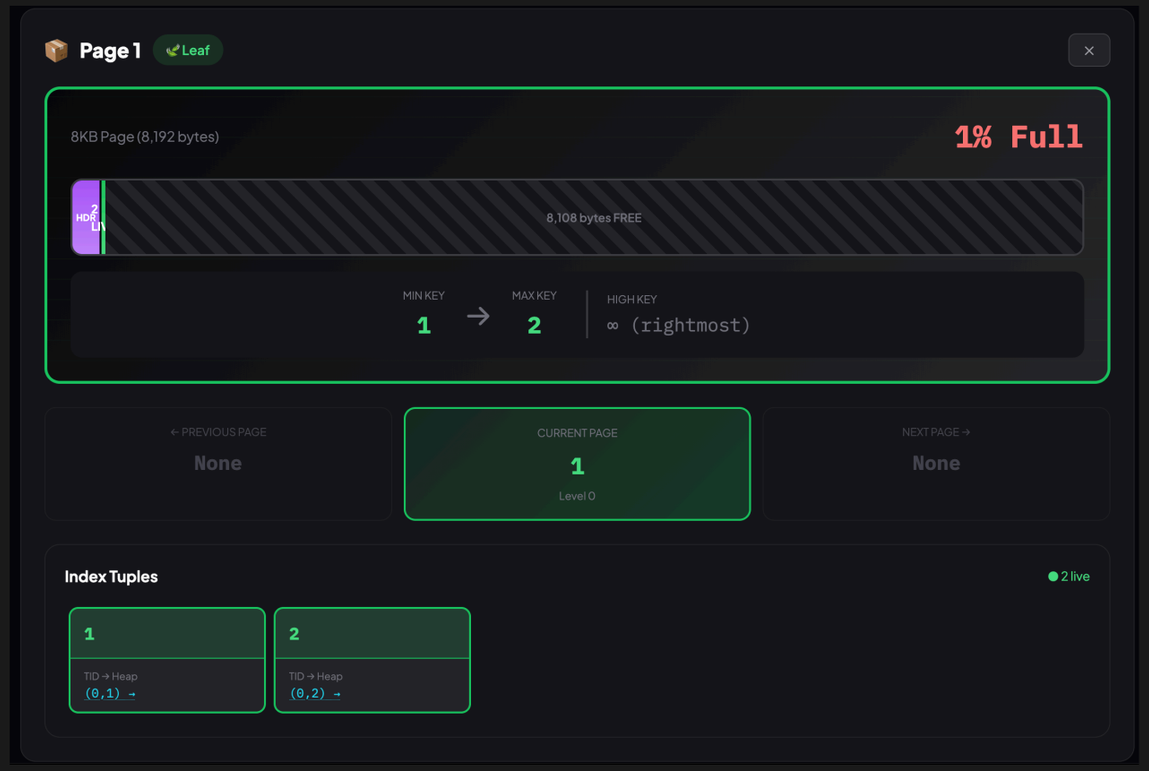
PG DATA 2026

boringSQL

Heap Page vs B-tree Page

Same 8KB skeleton, different cargo

	Heap Page	B-tree Page
Page header	24B, identical	24B, identical
Line pointers	point to heap tuples	point to index tuples
Stored data	full row (tuple)	key + ctid
Special space	**empty**	**sibling + level pointers**
MVCC	xmin / xmax per tuple	none — lives in the heap



Key difference: special space at the bottom holds B-tree navigation pointers (left page, right page, tree level).

boringSQL

What's in an index tuple?

Simpler version of a heap tuple

The **indexed column value** (the key)

A **ctid** pointing back to the heap row: `(page, line_pointer)`

No xmin/xmax **-- indexes don't do MVCC directly**

Indexes point to ALL versions of a row. Dead or alive.

> *Index entries have no xmin/xmax — visibility lives in the heap*

boringSQL

Not every UPDATE bloats the index

The one the index never hears about

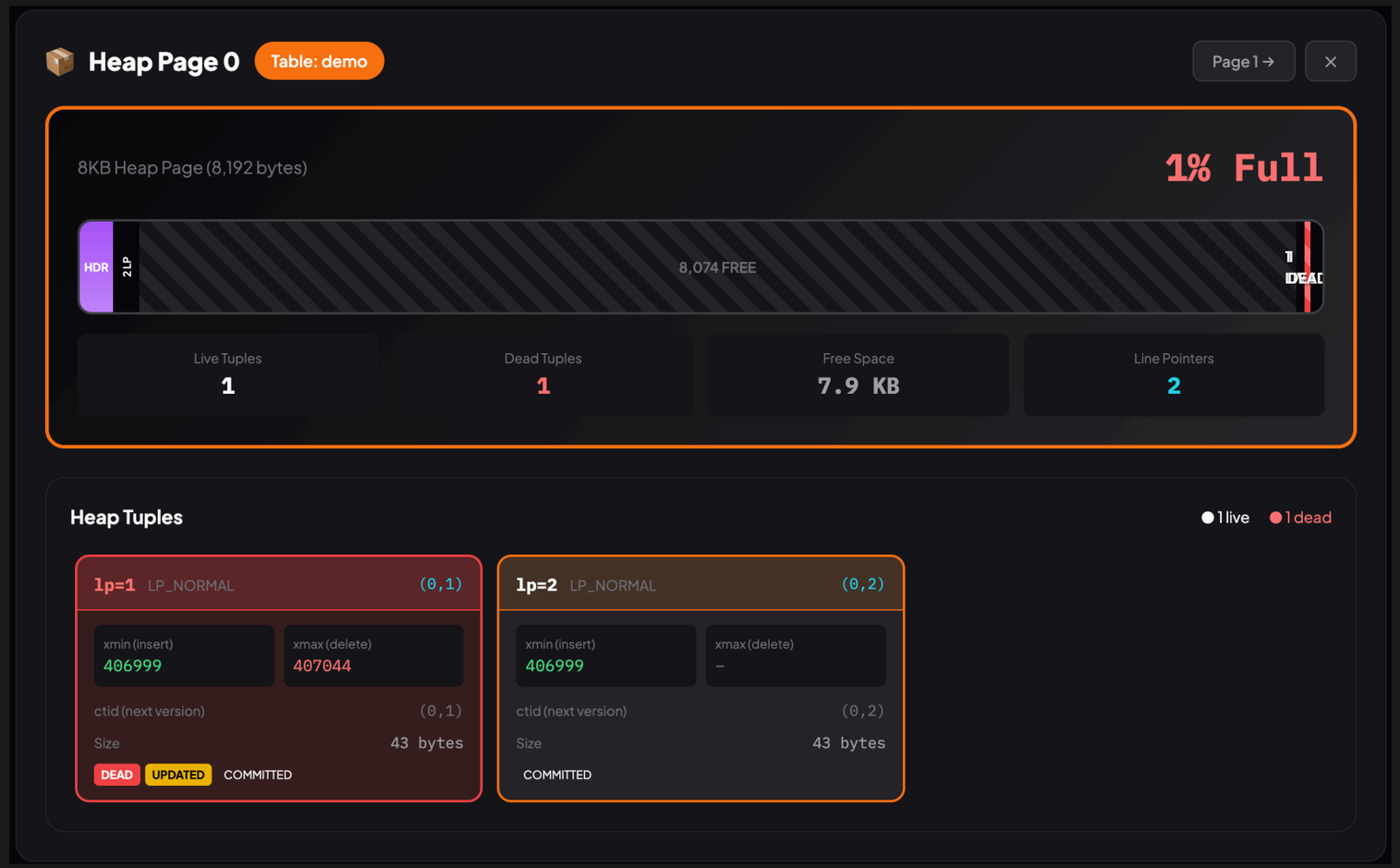
...with room left on the heap page

→ a **HOT** update (Heap-Only Tuple)

The index gets **no new entry**. None. WIN!

Don't just throw indexes left and right.

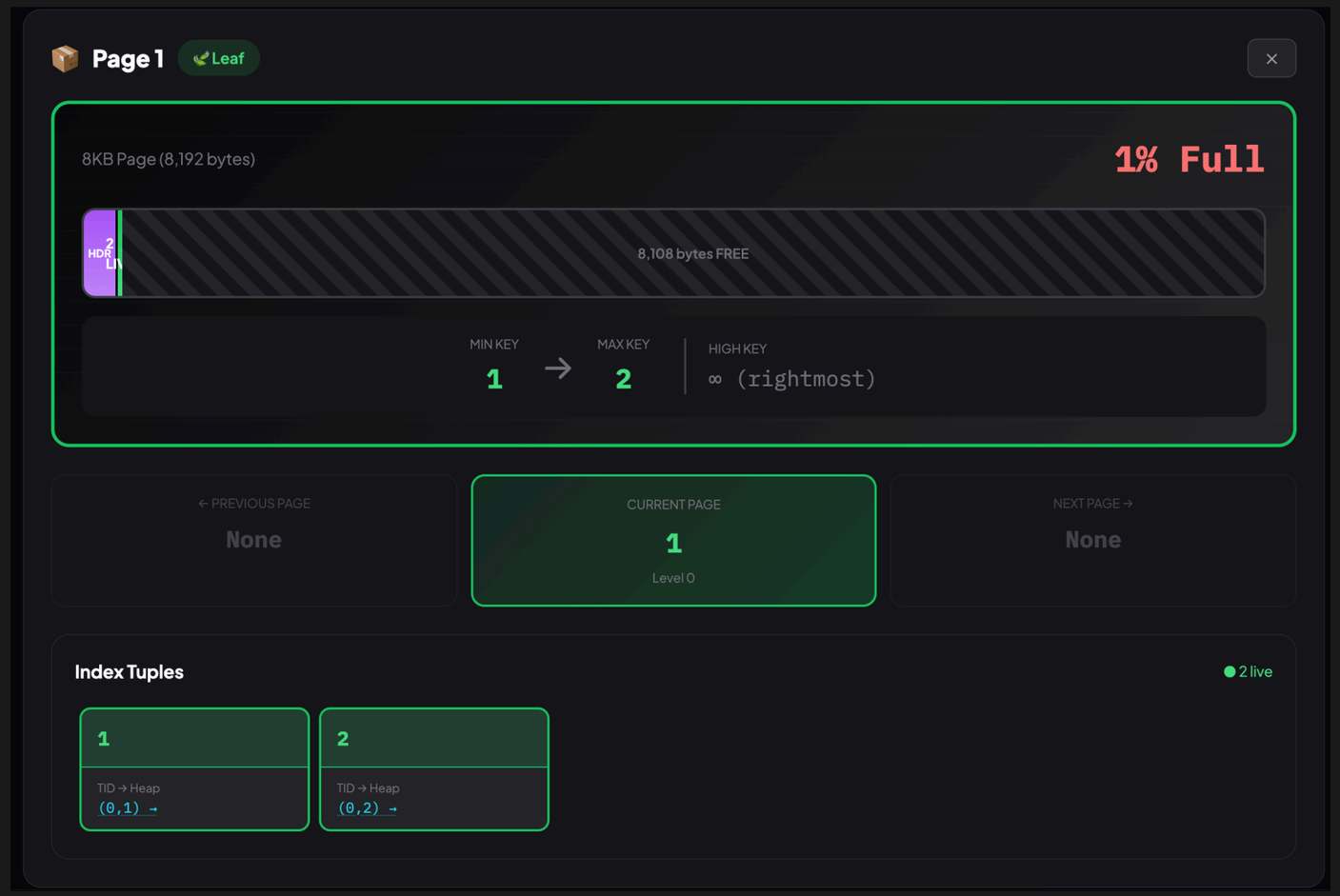
boringSQL



Heap Page

PG DATA 2026

boringSQL



B-Tree Page

PG DATA 2026

boringSQL

VACUUM

The False hope for indexes

PG DATA 2026

boringSQL

Why VACUUM can't compact B-tree indexes

1. B-tree pages are **linked in order** (left-right sibling pointers)
2. Removing a page from the middle means **relinking siblings**
3. Other transactions might be **mid-scan** through those pages
4. PostgreSQL takes the safe path: **leave the page, mark it reusable**

REINDEX

The real fix

```
REINDEX INDEX demo_pkey;

SELECT * FROM pgstatindex('demo_pkey');
```

How to stay "clean"

For table it's simple

```
ALTER TABLE demo SET (
    autovacuum_vacuum_scale_factor = 0.01,
    autovacuum_analyze_scale_factor = 0.01
);
```

For index "it depends"

```
SELECT
    schemaname || '.' || indexrelname as index,
    pg_size_pretty(pg_relation_size(indexrelid)) as size,
    idx_scan as scans,
    idx_tup_read as tuples_read,
    idx_tup_fetch as tuples_fetched
FROM pg_stat_user_indexes
ORDER BY pg_relation_size(indexrelid) DESC;
```

Red flags

Index size >> table size

`avg_leaf_density` below 50%

Large `empty_pages` or `deleted_pages` counts

Index size keeps growing despite stable row count

Problem	Fix	Downtime?
Index bloat	`REINDEX CONCURRENTLY`	No (PG 12+)
Index bloat (older PG)	`CREATE INDEX CONCURRENTLY` + swap	No
Index bloat (urgent)	`REINDEX`	**Yes** (exclusive lock)
Heap bloat	`VACUUM FULL` or `pg_repack`	Depends
Prevent bloat	Tune autovacuum per table	No

What VACUUM does vs. what it doesn't

	Heap	B-tree Index
Remove dead tuples	**Yes**	**Yes** (entries)
Reclaim space within page	**Yes**	**Yes**
Return pages to OS	Only trailing pages	**No**
Compact/defragment	Page-level pruning	**No**
Rebuild structure	No	**No**

VACUUM is necessary but not sufficient for indexes.

github.com/boringSQL/pg-storage-visualizer

PG DATA 2026

boringSQL

boringsql.com

https://boringsql.com/visualizers/

https://github.com/boringSQL/

boringSQL

DEMO TIME

Easiest way to fail live

PG DATA 2026

boringSQL

Feedback appreciated

Visualizing PostgreSQL Storage Internals

boringSQL