

Visualizing PostgreSQL Storage Internals

boringsQL

radim@boringsql.com
Skeptic, Turned Believer

2026.pgconf.de

boringsQL

Why should you care?

This is not academic knowledge

Table bloat

VACUUM

Performance

Concurrency

DELETES and UPDATES

The toolkit

The pg-storage-visualizer

github.com/boringSQL/pg-storage-visualizer

pageinspect

pgstattuple

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.

```
SHOW block_size;
```

```
block_size
```

```
-----
```

```
8192
```



Heap Page 0

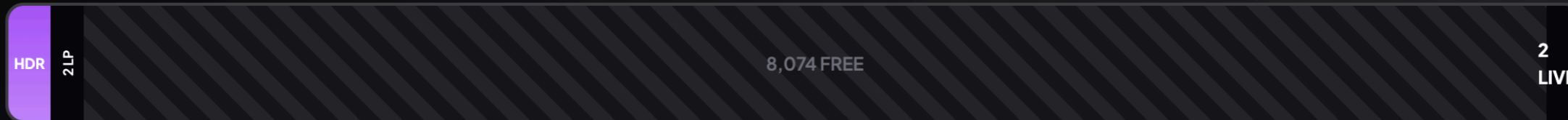
Table: demo

Page 1 →



8KB Heap Page (8,192 bytes)

1% Full



Live Tuples

2

Dead Tuples

0

Free Space

7.9 KB

Line Pointers

2

Heap Tuples

● 2 live

lp=1 LP_NORMAL

(0,1)

xmin (insert)

406999

xmax (delete)

-

ctid (next version)

(0,1)

Size

43 bytes

COMMITTED

lp=2 LP_NORMAL

(0,2)

xmin (insert)

406999

xmax (delete)

-

ctid (next version)

(0,2)

Size

43 bytes

COMMITTED

Page Header Line Pointers Data

Every row carries 23 bytes of overhead

HeapTupleHeaderData struct

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

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
```

The B-tree Index Page

Same 8KB. Different layout

Page 1 Leaf ×

8KB Page (8,192 bytes) **1% Full**

HDR 2
LV

8,108 bytes FREE

MIN KEY **1** → MAX KEY **2** | HIGH KEY ∞ (rightmost)

← PREVIOUS PAGE **None** | CURRENT PAGE **1** Level 0 | NEXT PAGE → **None**

Index Tuples ● 2 live

1	2
TID → Heap (0,1) →	TID → Heap (0,2) →

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

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*





Heap Page 0

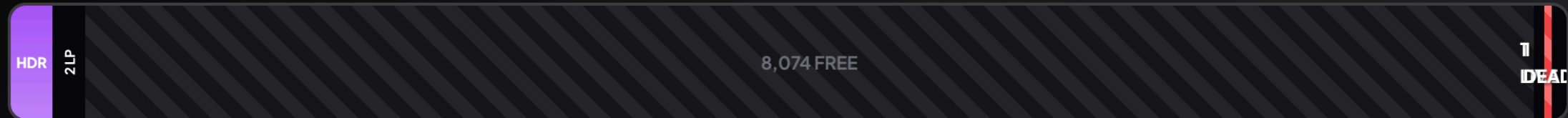
Table: demo

Page 1 →



8KB Heap Page (8,192 bytes)

1% Full



Live Tuples

1

Dead Tuples

1

Free Space

7.9 KB

Line Pointers

2

Heap Tuples

● 1 live ● 1 dead

lp=1 LP_NORMAL (0,1)

xmin (insert)

406999

xmax (delete)

407044

ctid (next version)

(0,1)

Size

43 bytes

DEAD

UPDATED

COMMITTED

lp=2 LP_NORMAL (0,2)

xmin (insert)

406999

xmax (delete)

-

ctid (next version)

(0,2)

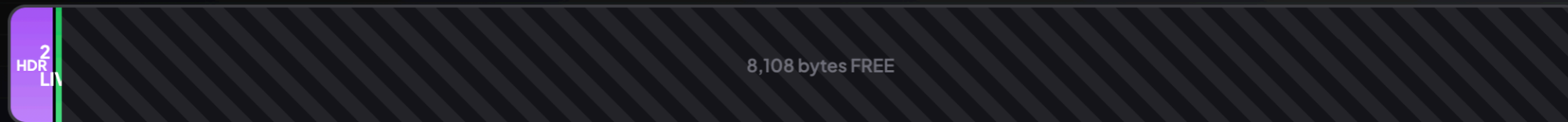
Size

43 bytes

COMMITTED

8KB Page (8,192 bytes)

1% Full



MIN KEY → MAX KEY | HIGH KEY
1 → **2** | ∞ (rightmost)

← PREVIOUS PAGE

None

CURRENT PAGE

1

Level 0

NEXT PAGE →

None

Index Tuples

● 2 live

1	2
TID → Heap (0,1) →	TID → Heap (0,2) →

VACUUM

The False hope for indexes

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	<code>REINDEX CONCURRENTLY</code>	No (PG 12+)
Index bloat (older PG)	<code>CREATE INDEX CONCURRENTLY</code> + swap	No
Index bloat (urgent)	<code>REINDEX</code>	Yes (exclusive lock)
Heap bloat	<code>VACUUM FULL</code> or <code>pg_repack</code>	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

boringsql.com

<https://boringsql.com/visualizers/>

<https://github.com/boringSQL/>

DEMO TIME

Easiest way to fail live