Solving Complex Database Problems by Starting Small
A Single-Page PostgreSQL Internals Deep-Dive

Talk: PGConf.DE 2026 – April 22, 2026
Table used: `onepage(id bigint, t text, d date)`
Environment: PostgreSQL 18.3, block_size = 8192, `pageinspect` extension installed

---

Setup

Before any demo, we disable autovacuum on the table so it does not interfere with our observations, and we establish two reusable helper queries.

```sql
-- Disable autovacuum so it does not clean up behind us
ALTER TABLE onepage SET (autovacuum_enabled = false);

-- Establish row sizing: repeat('x', 700) produces a 704-byte varlena column.
-- Each full tuple is 744 bytes (24 header + 8 id + 704 text + 4 date + 4 alignment padding),
-- aligned to 8 bytes. With a 4-byte line pointer, each row consumes 748 bytes.
-- An 8192-byte page holds at most 10 such rows:
--   page header: 24 bytes
--   10 line pointers: 40 bytes  → pd_lower = 64
--   10 × 744 bytes of tuple data  → pd_upper = 8192 - 7440 = 752
--   free space: 752 - 64 = 688 bytes (not enough for an 11th row)
```

Page inspection helpers (used throughout all demos):

```sql
-- Page header: shows pd_lower, pd_upper, pd_prune_xid
SELECT * FROM page_header(get_raw_page('onepage', 0));

-- Line pointer + tuple detail
SELECT
  lp          AS slot,
  lp_flags,
  t_xmin,
  t_xmax,
  t_ctid,
  lp_off,
  lp_len,
  CASE lp_flags
    WHEN 0 THEN 'unused'
    WHEN 1 THEN 'normal'
    WHEN 2 THEN 'redirect'
    WHEN 3 THEN 'dead'
  END AS flag_desc
```

```
FROM heap_page_items(get_raw_page('onepage', 0))
ORDER BY lp;
```

Section 1 — xmin, xmax, and Transaction Status

Why this matters

Every heap tuple carries two hidden system columns: `xmin` (the transaction ID that created it) and `xmax` (the transaction ID that deleted or replaced it). Together they define the tuple's visibility window under MVCC.

1a — Inserting rows: xmin is set, xmax = 0

```sql
TRUNCATE onepage;

INSERT INTO onepage VALUES (1, repeat('x', 700), '2024-01-01');
INSERT INTO onepage VALUES (2, repeat('x', 700), '2024-01-02');
INSERT INTO onepage VALUES (3, repeat('x', 700), '2024-01-03');

SELECT id, xmin, xmax, ctid FROM onepage ORDER BY id;
```

```
 id | xmin | xmax | ctid
----+------+------+-------
  1 |  770 |    0 | (0,1)
  2 |  771 |    0 | (0,2)
  3 |  772 |    0 | (0,3)
```

Each row has its own `xmin` (each INSERT was a separate autocommit transaction). `xmax = 0` means "no deletion pending" — this row is alive.

1b — Updating a row: the old version gets xmax set

```sql
UPDATE onepage SET d = '2024-06-01' WHERE id = 1;
```

A regular `SELECT` only shows the live (new) version:

```sql
SELECT id, xmin, xmax, ctid FROM onepage ORDER BY id;
```

```
 id | xmin | xmax | ctid
----+------+------+-------
```

```
  1 |  773 |    0 | (0,4)    ← new version, xmin = update txid
  2 |  771 |    0 | (0,2)
  3 |  772 |    0 | (0,3)
```

But `pageinspect` shows ALL versions — including the dead old row at slot 1:

```sql
SELECT
  lp AS slot,
  lp_flags,
  t_xmin,
  t_xmax,
  t_ctid,
  CASE lp_flags
    WHEN 0 THEN 'unused'
    WHEN 1 THEN 'normal'
    WHEN 2 THEN 'redirect'
    WHEN 3 THEN 'dead'
  END AS flag_desc
FROM heap_page_items(get_raw_page('onepage', 0))
ORDER BY lp;
```

```
 slot | lp_flags | t_xmin | t_xmax | t_ctid | flag_desc
------+----------+--------+--------+--------+-----------
    1 |        1 |    770 |    773 | (0,4)  | normal      ← dead old version (xmax set, points to new)
    2 |        1 |    771 |      0 | (0,2)  | normal
    3 |        1 |    772 |      0 | (0,3)  | normal
    4 |        1 |    773 |      0 | (0,4)  | normal      ← new live version of id=1
```

Key observations:
- Slot 1 (old version of `id=1`): `t_xmax=773` (the UPDATE transaction), `t_ctid=(0,4)` — the forward pointer to the new version.
- Slot 4 (new version): `t_xmin=773`, `t_ctid=(0,4)` (self-referential = no further version).

1c — Checking transaction status with pg_xact_status (commit log)

```sql
SELECT
  lp AS slot,
  t_xmin,
  t_xmax,
  t_ctid,
  pg_xact_status(t_xmin::text::xid8) AS xmin_status,
  CASE WHEN t_xmax = 0 THEN 'none'
```

```
       ELSE pg_xact_status(t_xmax::text::xid8)::text
  END AS xmax_status
FROM heap_page_items(get_raw_page('onepage', 0))
WHERE lp_flags = 1
ORDER BY lp;
```

```
 slot | t_xmin | t_xmax | t_ctid | xmin_status | xmax_status
------+--------+--------+--------+-------------+-------------
    1 |    830 |    833 | (0,4)  | committed   | committed   ← old
version of id=1 (dead: both sides committed)
    2 |    831 |      0 | (0,2)  | committed   | none        ← id=2,
live
    3 |    832 |      0 | (0,3)  | committed   | none        ← id=3,
live
    4 |    833 |      0 | (0,4)  | committed   | none        ← new
version of id=1, live
```

Slot 1 is the dead old version of `id=1`: `xmin_status=committed` (it was a valid insert) and `xmax_status=committed` (the UPDATE that replaced it is committed). Any snapshot newer than XID 833 will skip this tuple. Slots 2–4 have `xmax_status=none` (`xmax=0`) meaning no deletion is pending – they are alive.

1d – An aborted transaction leaves a dead xmax

```sql
BEGIN;
UPDATE onepage SET d = '2025-12-31' WHERE id = 2;
SELECT txid_current();   -- captures the XID before rollback
ROLLBACK;
```

```
 txid_current
--------------
          774
```

After the rollback, the page still has a "ghost" xmax on slot 2. A new read of the page will see `xmax_status = aborted` and know to ignore it – the row is still alive.

```sql
SELECT
  lp AS slot, t_xmin, t_xmax, t_ctid,
  pg_xact_status(t_xmin::text::xid8) AS xmin_status,
  CASE WHEN t_xmax = 0 THEN 'none'
       ELSE pg_xact_status(t_xmax::text::xid8)::text
  END AS xmax_status
FROM heap_page_items(get_raw_page('onepage', 0))
WHERE lp_flags = 1
```

ORDER BY lp;
```

```
 slot | t_xmin | t_xmax | t_ctid | xmin_status | xmax_status
------+--------+--------+--------+-------------+-------------
    1 |    770 |    773 | (0,4)  | committed   | committed   ← old
version of id=1 (genuinely dead)
    2 |    771 |    774 | (0,2)  | committed   | aborted     ← id=2:
xmax is ABORTED → row is still alive
    3 |    772 |      0 | (0,3)  | committed   | none
    4 |    773 |      0 | (0,4)  | committed   | none
    5 |    774 |      0 | (0,5)  | aborted     | none        ← ghost
tuple from the rolled-back UPDATE
```

Takeaways:
- `xmax = 0` → row is live, no deletion pending.
- `xmax` pointing to a **committed** transaction → row is genuinely dead.
- `xmax` pointing to an **aborted** transaction → PostgreSQL ignores it; row is alive.
- `xmin` from an **aborted** transaction (slot 5) → the tuple itself was never committed; invisible to all.

Section 2 — Hint Bits: Lazy Commit Status Caching

Why this matters

Checking whether `xmin` or `xmax` is committed requires reading the transaction status from the commit log (CLOG/pg_xact). This is a disk I/O per tuple on every visibility check — very expensive. PostgreSQL caches the result directly in the tuple's `t_infomask` field as **hint bits**, set lazily on the first read after commit.

Key infomask bits

| Hex | Name | Meaning
|
|--------|--------------------|-------------------------------------
------------|
| 0x0002 | HEAP_HASVARWIDTH | Tuple contains variable-width
columns |
| 0x0100 | HEAP_XMIN_COMMITTED| xmin's transaction committed (hint
bit) |
| 0x0200 | HEAP_XMIN_INVALID | xmin's transaction aborted/invalid
(hint bit) |
| 0x0400 | HEAP_XMAX_COMMITTED| xmax's transaction committed (hint
bit) |
| 0x0800 | HEAP_XMAX_INVALID | xmax is invalid / no valid deletion
(hint bit) |
| 0x1000 | HEAP_UPDATED | This tuple is the result of an
```

```
UPDATE           |
```

2a — Right after INSERT/COMMIT: XMIN_COMMITTED is NOT yet set

```sql
TRUNCATE onepage;
INSERT INTO onepage VALUES (1, repeat('x', 700), '2024-01-01');
-- Do NOT select from the table yet — we check the raw page first
```

```sql
SELECT
  lp AS slot,
  t_xmin,
  t_infomask,
  to_hex(t_infomask) AS infomask_hex,
  (t_infomask & x'0100'::int) != 0 AS xmin_committed_hint,
  (t_infomask & x'0800'::int) != 0 AS xmax_invalid_hint
FROM heap_page_items(get_raw_page('onepage', 0))
ORDER BY lp;
```

```
 slot | t_xmin | t_infomask | infomask_hex | xmin_committed_hint |
xmax_invalid_hint
------+--------+------------+--------------+---------------------
+------------------
    1 |    776 |       2050 | 802          | f                   | t
```

`0x802 = 0x800 | 0x002` → `XMAX_INVALID` (trivially: xmax=0) and `HASVARWIDTH`. **`XMIN_COMMITTED` is false** — PostgreSQL has not yet confirmed the commit by setting the hint bit.

2b — After a SELECT: XMIN_COMMITTED is now set

```sql
SELECT id, d FROM onepage;   -- seqscan reads the page, sets hint
bits as a side-effect
```

```
 id |     d
----+------------
  1 | 2024-01-01
```

```sql
SELECT
  lp AS slot,
  t_xmin,
  t_infomask,
  to_hex(t_infomask) AS infomask_hex,
  (t_infomask & x'0100'::int) != 0 AS xmin_committed_hint,
```

```
  (t_infomask & x'0800'::int) != 0 AS xmax_invalid_hint
FROM heap_page_items(get_raw_page('onepage', 0))
ORDER BY lp;
```

```
 slot | t_xmin | t_infomask | infomask_hex | xmin_committed_hint |
xmax_invalid_hint
------+--------+------------+--------------+---------------------
+------------------
    1 |    776 |       2306 | 902          | t                   | t
```

`0x902 = 0x800 | 0x100 | 0x002` → `XMAX_INVALID`,
`XMIN_COMMITTED` (now set!), `HASVARWIDTH`. The SELECT read the
tuple, confirmed the commit via CLOG, and wrote the result into
`t_infomask`. All future readers can skip the CLOG lookup.

2c — Full infomask decoder for a deleted row

```sql
INSERT INTO onepage VALUES (2, repeat('x', 700), '2024-01-02');
DELETE FROM onepage WHERE id = 1;
SELECT id FROM onepage;   -- seqscan to set hint bits

SELECT
  lp AS slot,
  t_xmin, t_xmax,
  to_hex(t_infomask) AS infomask_hex,
  ARRAY_REMOVE(ARRAY[
    CASE WHEN (t_infomask & x'0002'::int) != 0 THEN 'HAS_VARWIDTH'
END,
    CASE WHEN (t_infomask & x'0100'::int) != 0 THEN 'XMIN_COMMITTED'
END,
    CASE WHEN (t_infomask & x'0200'::int) != 0 THEN 'XMIN_INVALID'
END,
    CASE WHEN (t_infomask & x'0400'::int) != 0 THEN 'XMAX_COMMITTED'
END,
    CASE WHEN (t_infomask & x'0800'::int) != 0 THEN 'XMAX_INVALID'
END,
    CASE WHEN (t_infomask & x'1000'::int) != 0 THEN 'UPDATED'
END
  ], NULL) AS hint_bits_set
FROM heap_page_items(get_raw_page('onepage', 0))
ORDER BY lp;
```

```
 slot | t_xmin | t_xmax | infomask_hex |
hint_bits_set
------+--------+--------+--------------
+-----------------------------------------------
    1 |    776 |    778 | 502          |
{HAS_VARWIDTH,XMIN_COMMITTED,XMAX_COMMITTED}
```

```
    2 |    777 |      0 | 902          |
{HAS_VARWIDTH,XMIN_COMMITTED,XMAX_INVALID}
```

- Slot 1 (deleted): both `XMIN_COMMITTED` and `XMAX_COMMITTED` → tuple was created and then definitively deleted. Dead.
- Slot 2 (live): `XMIN_COMMITTED` and `XMAX_INVALID` → created by a committed transaction, no deletion. Alive.

Takeaway: Hint bits are a write-on-first-read optimisation. The first reader to touch a committed tuple pays the CLOG cost and then stamps the answer into the tuple for everyone else. This is why a fresh INSERT on an idle server shows `XMIN_COMMITTED = false` until someone reads that row.

Section 3 — Single Page Cleanup (Heap Pruning)

Why this matters

When old tuple versions accumulate, they waste page space and slow sequential scans. PostgreSQL can reclaim that space **without a full VACUUM** through a mechanism called **heap page pruning**. Pruning is triggered when a heap page scan (seqscan or the page check before an INSERT) detects the page's `prune_xid` is older than the current global transaction horizon.

3a — Fill half a page, then update rows to create dead versions

```sql
TRUNCATE onepage;
-- Drop any index so updates use index scan paths, not seqscan
DROP INDEX IF EXISTS onepage_id_idx;

INSERT INTO onepage SELECT g, repeat('x', 700), '2024-01-01'::date
FROM generate_series(1, 5) g;
```

```sql
-- 5 rows, page has ~3700 bytes free (room for 3 more rows + their
new versions)
SELECT * FROM page_header(get_raw_page('onepage', 0));
```

```
    lsn    | lower | upper | prune_xid
-----------+-------+-------+-----------
 0/1861CB8 |    44 |  4472 |         0
```

`upper=4472`, `lower=44` → free space = 4428 bytes. Enough for both old dead versions and new live versions to coexist on the same page.

```sql
-- Create an index so UPDATEs use index scan (avoids seqscan
triggering pruning prematurely)
CREATE INDEX ON onepage(id);

UPDATE onepage SET d = '2025-06-01' WHERE id IN (1, 2, 3);
```

3b — Inspect page IMMEDIATELY after UPDATE: dead tuples are visible

```sql
SELECT * FROM page_header(get_raw_page('onepage', 0));
```

```
    lsn    | lower | upper | prune_xid
-----------+-------+-------+-----------
 0/1861DF0 |    56 |  2240 |       791
```

`prune_xid=791` — the page itself knows that transaction 791 left dead tuples behind. The next eligible page scan can clean them up.

```sql
SELECT
  lp AS slot,
  lp_flags,
  t_xmin, t_xmax, t_ctid,
  lp_off, lp_len,
  CASE
    WHEN lp_flags = 1 AND t_xmax::text != '0' THEN 'DEAD TUPLE (xmax
set, data still on page)'
    WHEN lp_flags = 1 AND t_xmax::text = '0' THEN 'live'
  END AS status
FROM heap_page_items(get_raw_page('onepage', 0))
ORDER BY lp;
```

```
 slot | lp_flags | t_xmin | t_xmax | t_ctid | lp_off | lp_len |
status
------+----------+--------+--------+--------+--------+--------
+---------------------------------------
    1 |        1 |    790 |    791 | (0,6)  |   7448 |    740 | DEAD
TUPLE (xmax set, data still on page)
    2 |        1 |    790 |    791 | (0,7)  |   6704 |    740 | DEAD
TUPLE (xmax set, data still on page)
    3 |        1 |    790 |    791 | (0,8)  |   5960 |    740 | DEAD
TUPLE (xmax set, data still on page)
    4 |        1 |    790 |      0 | (0,4)  |   5216 |    740 | live
    5 |        1 |    790 |      0 | (0,5)  |   4472 |    740 | live
    6 |        1 |    791 |      0 | (0,6)  |   3728 |    740 | live
← new version of id=1
```

```
    7 |        1 |    791 |      0 | (0,7)  |   2984 |    740 | live
← new version of id=2
    8 |        1 |    791 |      0 | (0,8)  |   2240 |    740 | live
← new version of id=3
```

The old versions of rows 1–3 (slots 1–3) are dead: `t_xmax=791` (committed) and `t_ctid` pointing forward to the new versions at slots 6–8. Their tuple data (`lp_len=740`) is **still physically on the page**, taking up ~2220 bytes of wasted space.

3c — A seqscan from a fresh connection triggers pruning (no VACUUM needed)

```sql
-- Run from a NEW psql connection (fresh global transaction horizon)
SELECT COUNT(*) FROM onepage;
```

```
 count
-------
     5
```

Now inspect the page again:

```sql
SELECT * FROM page_header(get_raw_page('onepage', 0));
```

```
    lsn    | lower | upper | prune_xid
-----------+-------+-------+-----------
 0/18667E0 |    64 |  2984 |         0
```

The LSN changed (page was written), `prune_xid` is back to 0, and `upper` jumped from 2240 to 2984 — ~744 bytes of dead tuple data freed for each pruned version.

```sql
SELECT
  lp AS slot, lp_flags, t_xmin, t_xmax, t_ctid, lp_off, lp_len,
  CASE lp_flags
    WHEN 0 THEN 'unused'
    WHEN 1 THEN 'normal'
    WHEN 2 THEN 'redirect (HOT chain head)'
    WHEN 3 THEN 'dead line pointer'
  END AS flag_desc
FROM heap_page_items(get_raw_page('onepage', 0))
ORDER BY lp;
```

```
 slot | lp_flags | t_xmin | t_xmax | t_ctid | lp_off | lp_len |
flag_desc
------+----------+--------+--------+--------+--------+--------
+-------------------------
    1 |        2 |        |        |        |      6 |      0 |
redirect (HOT chain head)
    2 |        2 |        |        |        |      7 |      0 |
redirect (HOT chain head)
    3 |        2 |        |        |        |      8 |      0 |
redirect (HOT chain head)
    4 |        1 |    790 |      0 | (0,4)  |   7448 |    740 |
normal
    5 |        1 |    790 |      0 | (0,5)  |   6704 |    740 |
normal
    6 |        1 |    791 |      0 | (0,6)  |   5960 |    740 |
normal
    7 |        1 |    791 |      0 | (0,7)  |   5216 |    740 |
normal
    8 |        1 |    791 |      0 | (0,8)  |   4472 |    740 |
normal
```

The dead old versions (slots 1–3) have been transformed:
- Their tuple **data is freed** (`lp_len=0`, `upper` moved upward).
- The line pointers became **redirects** (`lp_flags=2`): slot 1 → slot 6, slot 2 → slot 7, slot 3 → slot 8. This preserves the HOT chain linkage — any reader following ctid `(0,1)` is forwarded to the live tuple at `(0,6)`.
- No VACUUM was involved. A regular `SELECT COUNT(*)` was sufficient.

Important note on when pruning fires: Pruning requires that the page's `prune_xid` is below the global transaction horizon (`GetOldestNonRemovableTransactionId()`). In a production system with many concurrent sessions, this condition is satisfied almost immediately after commit. In a single-session demo environment, it requires a fresh psql connection whose snapshot is newer than the dead transactions. VACUUM always triggers pruning regardless.

Section 3.5 — Does PostgreSQL Go Back to Insert at the Start of a Page?

The question

After pruning freed ~2220 bytes on page 0, will new `INSERT`s reuse that space, or will PostgreSQL keep extending the table file?

The Free Space Map (FSM)

PostgreSQL tracks available space per page in a side structure called the **Free Space Map**. When `RelationGetBufferForTuple`

looks for a page to insert into, it consults the FSM. If the FSM doesn't know about the freed space, new inserts will extend the file.

The FSM is updated by:
1. **VACUUM** — reliably after every run.
2. **Heap pruning** — only writes back to FSM under certain conditions; not guaranteed in all pruning paths.

3.5a — Insert after pruning (without VACUUM): FSM is stale → goes to page 1

```sql
INSERT INTO onepage VALUES (200, repeat('z', 700), '2026-01-01');
INSERT INTO onepage VALUES (201, repeat('z', 700), '2026-01-01');
INSERT INTO onepage VALUES (202, repeat('z', 700), '2026-01-01');

SELECT id, ctid FROM onepage WHERE id >= 200 ORDER BY ctid;
```

```
 id  |  ctid
-----+--------
 200 | (1,2)
 201 | (1,3)
 202 | (1,4)
```

All three went to **page 1**. The FSM still records page 0 as full (the pruning path in this case did not update the FSM). PostgreSQL extended the file rather than going back.

3.5b — After VACUUM updates the FSM: inserts return to page 0

```sql
VACUUM onepage;   -- regular (non-FULL) VACUUM refreshes the FSM

INSERT INTO onepage VALUES (300, repeat('z', 700), '2026-06-01');

SELECT id, ctid FROM onepage WHERE id = 300;
```

```
 id  |  ctid
-----+-------
 300 | (0,11)
```

Row 300 went to **page 0, slot 11** — reusing the freed space that pruning reclaimed. PostgreSQL happily goes back to fill earlier pages once the FSM is up-to-date.

Answer: Yes, PostgreSQL will insert into freed space on an earlier page, but only when the Free Space Map knows about it.

VACUUM is the reliable mechanism for updating the FSM. In high-update workloads, the FSM stays current naturally because VACUUM runs frequently.

Section 4 — Fillfactor: How Many Rows Fit, and What Happens at the Boundary?

Why this matters

`fillfactor` tells PostgreSQL how full to pack a page during **inserts**. The remaining space is reserved for **in-place update versions** (HOT updates). The two key questions are:

1. How many rows fit before inserts spill to a new page?
2. Does PostgreSQL cross the fillfactor threshold, or does the row go to a new page cleanly?

4a — Setup: fillfactor = 75

```sql
DROP INDEX IF EXISTS onepage_id_idx;
ALTER TABLE onepage SET (fillfactor = 75);
TRUNCATE onepage;
```

With `fillfactor=75`, PostgreSQL reserves 25% of each page for updates. Threshold calculation:
- Available space: 8192 bytes
- Reserved: 8192 × 25% = 2048 bytes minimum free space required after each INSERT
- A new row needs: 748 bytes (4 LP + 744 tuple)
- After 7 rows: free = 8168 − (7 × 748) = 2932 bytes → check: 2932 − 748 = 2184 ≥ 2048 → row 8 fits
- After 8 rows: free = 8168 − (8 × 748) = 2184 bytes → check: 2184 − 748 = 1436 < 2048 → row 9 does NOT fit on page 0

4b — Insert rows one by one and observe the page boundary

```sql
INSERT INTO onepage SELECT g, repeat('x', 700), '2024-01-01'::date
FROM generate_series(1, 9) g;

SELECT id, ctid,
       (ctid::text::point)[0]::int AS page,
       (ctid::text::point)[1]::int AS slot
FROM onepage ORDER BY id;
```

```
 id | ctid  | page | slot
----+-------+------+------
  1 | (0,1) |    0 |    1
```

```
  2 | (0,2) |    0 |    2
  3 | (0,3) |    0 |    3
  4 | (0,4) |    0 |    4
  5 | (0,5) |    0 |    5
  6 | (0,6) |    0 |    6
  7 | (0,7) |    0 |    7
  8 | (0,8) |    0 |    8   ← last row that fits within
fillfactor=75
  9 | (1,1) |    1 |    1   ← next INSERT went to a NEW page —
threshold not crossed
```

```sql
-- Page 0 state after 8 rows
SELECT * FROM page_header(get_raw_page('onepage', 0));
```

```
    lsn    | lower | upper | prune_xid
-----------+-------+-------+-----------
 0/1877E60 |    56 |  2240 |         0
```

`upper=2240`, `lower=56` → free = 2184 bytes. This is the **reserved space** held back for future UPDATE versions.

Answer to the question: PostgreSQL does **not** cross the fillfactor threshold. Row 9 goes cleanly to a new page without squeezing into the reserved space. The 2184 reserved bytes on page 0 are available exclusively for UPDATE versions of rows 1–8 (particularly HOT updates).

4c — UPDATEs use the reserved space

```sql
CREATE INDEX onepage_id_idx ON onepage(id);
UPDATE onepage SET d = '2025-06-01' WHERE id = 1;   -- new version
goes to reserved space
UPDATE onepage SET d = '2025-06-02' WHERE id = 2;   -- second new
version fits too
UPDATE onepage SET d = '2025-06-03' WHERE id = 3;   -- third new
version overflows to page 1
```

```sql
SELECT * FROM page_header(get_raw_page('onepage', 0));
```

```
    lsn    | lower | upper | prune_xid | flags
-----------+-------+-------+-----------+-------
 0/188FCC0 |    64 |   752 |       803 |     2   ← PD_PAGE_FULL
```

```sql
SELECT lp, lp_flags, t_xmin, t_xmax, t_ctid, lp_off FROM
heap_page_items(get_raw_page('onepage', 0)) ORDER BY lp;
```

```
 lp | lp_flags | t_xmin | t_xmax | t_ctid | lp_off
----+----------+--------+--------+--------+--------
  1 |        1 |    801 |    803 | (0,9)  |   7448   ← old version
of id=1 (dead)
  2 |        1 |    801 |    803 | (0,10) |   6704   ← old version
of id=2 (dead)
  3 |        1 |    801 |    803 | (1,2)  |   5960   ← old version
of id=3 (dead, new ver → page 1)
  4 |        1 |    801 |      0 | (0,4)  |   5216   ← live
  ...
  9 |        1 |    803 |      0 | (0,9)  |   1496   ← new version
of id=1 (in reserved space!)
 10 |        1 |    803 |      0 | (0,10) |    752   ← new version
of id=2 (in reserved space!)
```

Updates 1 and 2 placed their new versions at slots 9 and 10 **within page 0** (using the reserved fillfactor space). Update 3 exhausted the reserved area and spilled to page 1. The `flags=2` marks the page as `PD_PAGE_FULL` — no more inserts will be attempted here.

Takeaway: `fillfactor` is not just a capacity knob. It is directly linked to HOT update efficiency. With `fillfactor=75`, each page can absorb ~2 UPDATE cycles before overflowing, keeping related versions co-located and enabling HOT chain traversal without cross-page I/O.

Section 5 — HOT Updates and Line Pointer Chains

Why this matters

A **Heap-Only Tuple (HOT)** update is one where:
1. No indexed column is changed, AND
2. The new tuple version fits on the same page as the old version.

When both conditions hold, PostgreSQL can insert the new version **without touching the index**. The old line pointer becomes a redirect to the new version. This is called a HOT chain.

5a — Setup

```sql
DROP INDEX IF EXISTS onepage_id_idx;
ALTER TABLE onepage SET (fillfactor = 75);
TRUNCATE onepage;
CREATE INDEX onepage_id_idx ON onepage(id);   -- index only on id,
```

NOT on t or d

INSERT INTO onepage SELECT g, repeat('v1', 350), '2024-01-01'::date
FROM generate_series(1, 5) g;
```

5 rows inserted. Page has ~4400 bytes free (fillfactor=75 reserves ~2000 for updates).

### 5b — First HOT update: 2-link chain (id=1, only t and d changed — no indexed column touched)

```sql
UPDATE onepage SET t = repeat('v2', 350), d = '2024-06-01' WHERE id = 1;
```

```sql
SELECT lp, lp_flags, t_xmin, t_xmax, t_ctid, lp_off, lp_len
FROM heap_page_items(get_raw_page('onepage', 0)) ORDER BY lp;
```

```
 lp | lp_flags | t_xmin | t_xmax | t_ctid | lp_off | lp_len
----+----------+--------+--------+--------+--------+--------
  1 |        1 |    808 |    809 | (0,6)  |   7448 |    740   ← dead old version, t_ctid → slot 6
  2 |        1 |    808 |      0 | (0,2)  |   6704 |    740
  3 |        1 |    808 |      0 | (0,3)  |   5960 |    740
  4 |        1 |    808 |      0 | (0,4)  |   5216 |    740
  5 |        1 |    808 |      0 | (0,5)  |   4472 |    740
  6 |        1 |    809 |      0 | (0,6)  |   3728 |    740   ← new live version of id=1
```

HOT chain: `(0,1)` → `(0,6)`. The index still points to slot 1 (unchanged). Any index scan for `id=1` goes to `(0,1)`, sees `lp_flags=1` with `t_ctid=(0,6)`, and follows the chain to the live version at slot 6. **No index update needed.**

5c — Verify the index still points to the original slot

```sql
SELECT itemoffset, ctid, data
FROM bt_page_items('onepage_id_idx', 1)
ORDER BY itemoffset;
```

```
 itemoffset | ctid  |          data
------------+-------+------------------------
          1 | (0,1) | 01 00 00 00 00 00 00 00   ← id=1 index entry: still (0,1)
          2 | (0,2) | 02 00 00 00 00 00 00 00
```
```

```
          3 | (0,3) | 03 00 00 00 00 00 00 00
          4 | (0,4) | 04 00 00 00 00 00 00 00
          5 | (0,5) | 05 00 00 00 00 00 00 00
```

The index has **not been updated**. It still points to `(0,1)`, which is now the dead head of a HOT chain.

5d — Second HOT update: 3-link chain

```sql
UPDATE onepage SET t = repeat('v3', 350), d = '2024-12-01' WHERE id = 1;
```

```sql
SELECT lp, lp_flags, t_xmin, t_xmax, t_ctid
FROM heap_page_items(get_raw_page('onepage', 0)) ORDER BY lp;
```

```
 lp | lp_flags | t_xmin | t_xmax | t_ctid
----+----------+--------+--------+--------
  1 |        1 |    808 |    809 | (0,6)   ← version 1 (dead): points to slot 6
  ...
  6 |        1 |    809 |    810 | (0,7)   ← version 2 (dead): points to slot 7
  7 |        1 |    810 |      0 | (0,7)   ← version 3 (LIVE): self-referential t_ctid
```

Chain: `slot 1 → slot 6 → slot 7 (live)`. The index entry `(0,1)` remains unchanged. PostgreSQL traverses the full chain on lookup.

5e — Chain compression during VACUUM: the middle link is removed

```sql
VACUUM onepage;
```

```sql
SELECT lp, lp_flags, t_xmin, t_xmax, t_ctid, lp_off, lp_len
FROM heap_page_items(get_raw_page('onepage', 0)) ORDER BY lp;
```

```
 lp | lp_flags | t_xmin | t_xmax | t_ctid | lp_off | lp_len
----+----------+--------+--------+--------+--------+--------
  1 |        2 |        |        |        |      7 |      0   ← REDIRECT: slot 1 now points directly to slot 7
  2 |        1 |    808 |      0 | (0,2)  |   7448 |    740
  3 |        1 |    808 |      0 | (0,3)  |   6704 |    740
  4 |        1 |    808 |      0 | (0,4)  |   5960 |    740
```

```
  5 |        1 |    808 |      0 | (0,5)  |   5216 |    740
  6 |        0 |        |        |        |      0 |      0   ←
UNUSED: dead middle link fully removed
  7 |        1 |    810 |      0 | (0,7)  |   4472 |    740   ← live
current version of id=1
```

```sql
-- Index still points to (0,1) — no index update was performed
SELECT itemoffset, ctid FROM bt_page_items('onepage_id_idx', 1)
ORDER BY itemoffset;
```

```
 itemoffset | ctid
------------+-------
          1 | (0,1)   ← index entry unchanged: (0,1) → now a
redirect → (0,7)
          2 | (0,2)
          ...
```

What VACUUM did to the HOT chain:

Before VACUUM	After VACUUM
Slot 1: normal, dead (data still present, 740 bytes)	Slot 1: **redirect** to slot 7 (no data, 0 bytes)
Slot 6: normal, dead (data still present, 740 bytes)	Slot 6: **unused** (completely freed)
Slot 7: live	Slot 7: live (unchanged)

The 3-link chain `1 → 6 → 7` was compressed to `1 (redirect) → 7`. The dead middle link (slot 6) is gone. The index entry `(0,1)` still works: it follows the redirect at slot 1 directly to the live tuple at slot 7 — no index vacuum needed.

Takeaways:
- HOT updates eliminate index maintenance overhead entirely as long as the non-indexed column is changed and the new version fits on the same page.
- `fillfactor < 100` is the prerequisite: without reserved space, the new version can't fit on the same page and HOT is impossible.
- Pruning (seqscan) can shorten live chains; VACUUM performs the final compression and marks the head as a pure redirect.
- VACUUM also sets `lp_flags=0` (unused) on dead middle-chain slots, making those line pointer slots available for reuse by future inserts.

Section 6 — Table Bloat vs. Dead Rows: What Is the Difference?

The confusion

These two terms are often used interchangeably, but they describe different problems:

Term	What it is	Visible to queries?	Reclaimed by	Returns space to OS?
Dead rows	Tuples with committed `xmax`, sitting on a page	No	Regular VACUUM	Only if whole pages become empty (rare)
Bloat	Pages allocated to the table that are mostly empty	n/a	VACUUM FULL / pg_repack	Yes — by rewriting the whole table

6a — State before cleanup: dead rows occupy physical space

```sql
TRUNCATE onepage;
ALTER TABLE onepage SET (fillfactor = 100);
INSERT INTO onepage SELECT g, repeat('x', 700), '2024-01-01'::date
FROM generate_series(1,10) g;

DELETE FROM onepage WHERE id <= 7;

-- Check file size immediately
SELECT pg_relation_size('onepage') AS file_size,
       pg_size_pretty(pg_relation_size('onepage')) AS size_pretty;
```

```
 file_size | size_pretty
-----------+-------------
      8192 | 8192 bytes
```

```sql
-- See dead rows via pageinspect (lp_flags=1, xmax set, lp_len=740: data still on page)
SELECT lp, lp_flags, t_xmin, t_xmax, lp_len FROM heap_page_items(get_raw_page('onepage', 0)) ORDER BY lp;
```

```
 lp | lp_flags | t_xmin | t_xmax | lp_len
----+----------+--------+--------+--------
  1 |        1 |    826 |    826 |    740   ← DEAD: xmin=xmax (created+deleted same txn)
  2 |        1 |    826 |    826 |    740   ← DEAD
  3 |        1 |    826 |    826 |    740   ← DEAD
  4 |        1 |    826 |    826 |    740   ← DEAD
  5 |        1 |    826 |    826 |    740   ← DEAD
  6 |        1 |    826 |    826 |    740   ← DEAD
  7 |        1 |    826 |    826 |    740   ← DEAD
  8 |        1 |    826 |      0 |    740   ← live
  9 |        1 |    826 |      0 |    740   ← live
```

```
 10 |        1 |    826 |      0 |    740   ← live
```

7 dead rows still physically occupy 7 × 740 = 5180 bytes on the page. The file is still 8192 bytes. Queries only see rows 8–10 but the wasted space affects seqscan performance (pages need to be read, then most tuples filtered).

6b — After regular VACUUM: dead rows removed, but file stays the same size

```sql
VACUUM (VERBOSE) onepage;

SELECT pg_relation_size('onepage') AS file_size_after_vacuum,
       pg_size_pretty(pg_relation_size('onepage')) AS size_pretty;
```

```
 file_size_after_vacuum | size_pretty
------------------------+-------------
                   8192 | 8192 bytes   ← UNCHANGED
```

```sql
SELECT lp, lp_flags, lp_off, lp_len FROM
heap_page_items(get_raw_page('onepage', 0)) ORDER BY lp;
```

```
 lp | lp_flags | lp_off | lp_len
----+----------+--------+--------
  1 |        0 |      0 |      0   ← unused (dead row freed)
  2 |        0 |      0 |      0   ← unused
  3 |        0 |      0 |      0   ← unused
  4 |        0 |      0 |      0   ← unused
  5 |        0 |      0 |      0   ← unused
  6 |        0 |      0 |      0   ← unused
  7 |        0 |      0 |      0   ← unused
  8 |        1 |   7448 |    740   ← live
  9 |        1 |   6704 |    740   ← live
 10 |        1 |   5960 |    740   ← live
```

```sql
SELECT * FROM page_header(get_raw_page('onepage', 0));
```

```
    lsn    | lower | upper | flags | prune_xid
-----------+-------+-------+-------+-----------
 0/18BC4A8 |    64 |  5960 |     5 |         0
```

`flags=5` = `PD_HAS_FREE_LINES (0x01) | PD_ALL_VISIBLE (0x04)`. The 7 line pointer slots are free (lp_flags=0) and the tuple data space is reclaimed (`upper` moved from 752 to 5960). The page now has 5924 bytes of free space available for new rows.

But the file is still 8192 bytes. PostgreSQL has reclaimed the space _inside_ the page but cannot shrink the file because:
1. The page still exists in the heap relation file.
2. Other pages (in a bigger table) might still be needed.
3. Shrinking the file requires a full rewrite.

6c — After VACUUM FULL: file is rewritten, dead slots removed

```sql
VACUUM FULL onepage;

SELECT pg_relation_size('onepage') AS file_size_after_vacuum_full,
       pg_size_pretty(pg_relation_size('onepage'));
```

```
 file_size_after_vacuum_full | size_pretty
-----------------------------+-------------
                        8192 | 8192 bytes   ← still 8192 (minimum is
1 page = 8192 bytes)
```

```sql
SELECT * FROM page_header(get_raw_page('onepage', 0));
```

```
    lsn    | lower | upper
-----------+-------+-------
 0/18E74A0 |    36 |  5960
```

`lower=36` = `24 + 3 × 4` — only **3 line pointers** remain. All 7 empty line pointer slots from before are gone. In a real table with many pages, VACUUM FULL would also release entirely empty pages, actually reducing `pg_relation_size`.

```sql
SELECT lp, lp_flags, lp_off, lp_len FROM
heap_page_items(get_raw_page('onepage', 0)) ORDER BY lp;
```

```
 lp | lp_flags | lp_off | lp_len
----+----------+--------+--------
  1 |        1 |   7448 |    740   ← live (was id=8)
  2 |        1 |   6704 |    740   ← live (was id=9)
  3 |        1 |   5960 |    740   ← live (was id=10)
```

Summary: Dead Rows vs. Bloat

```
            INSERT 10 rows → DELETE 7 rows
            ↓
  Page 0: [dead][dead][dead][dead][dead][dead][dead][live][live]
[live]
            ↑_______ 7 dead rows (5180 bytes wasted) _______↑
            File: 8192 bytes

            ↓ regular VACUUM
  Page 0: [free][free][free][free][free][free][free][live][live]
[live]
            ↑__ 5924 bytes free (available for new inserts) __↑
            File: 8192 bytes  ← BLOAT: allocated but underused

            ↓ VACUUM FULL (or pg_repack)
  Page 0: [live][live][live]
            ↑ 5924 bytes free ↑
            File: 8192 bytes (minimum — in a multi-page table this
would shrink)
```

Key distinctions:
- **Dead rows** (`n_dead_tup` in `pg_stat_user_tables`) = committed-deleted tuples whose data is still on the page. They cost I/O (pages still scanned) and space. Regular VACUUM reclaims their page space **for reuse within PostgreSQL** (the FSM learns about it).
- **Bloat** = pages that PostgreSQL keeps allocated but are mostly or entirely empty. Regular VACUUM cannot return these to the OS. `VACUUM FULL` (takes an `ACCESS EXCLUSIVE` lock, full table rewrite) or `pg_repack` (online, no full lock) are required to give the space back.
- In production, bloat builds up after heavy delete/update workloads followed by VACUUM. Monitoring `pg_stat_user_tables.n_dead_tup` catches the dead-rows problem early; monitoring `pg_relation_size` vs. actual row count catches bloat.

Section 7 — Table Bloat vs. Dead Rows: multi page example
```sql
create table multipage(like onepage including all);
```

Insert 10 rows and update all rows two times
```sql
INSERT INTO multipage SELECT g, repeat('x', 700), '2024-01-01'::date
FROM generate_series(1,10) g;
update multipage set d = '2025-01-01';
update multipage set d = '2026-01-01';

SELECT lp, lp_flags, t_xmin, t_xmax, lp_len FROM
```

```
heap_page_items(get_raw_page('multipage', 2)) ORDER BY lp;
 lp | lp_flags | t_xmin | t_xmax | lp_len
----+----------+--------+--------+--------
  1 |        1 |    850 |      0 |    740
  2 |        1 |    850 |      0 |    740
  3 |        1 |    850 |      0 |    740
  4 |        1 |    850 |      0 |    740
  5 |        1 |    850 |      0 |    740
  6 |        1 |    850 |      0 |    740
  7 |        1 |    850 |      0 |    740
  8 |        1 |    850 |      0 |    740
  9 |        1 |    850 |      0 |    740
 10 |        1 |    850 |      0 |    740

 SELECT * FROM page_header(get_raw_page('multipage', 0));
    lsn    | checksum | flags | lower | upper | special | pagesize |
version | prune_xid
-----------+----------+-------+-------+-------+---------+----------
+---------+-----------
 0/196C490 |        0 |     0 |    64 |  8192 |    8192 |     8192 |
4 |         0
 ```

A select will already clean up page 0 and 1
```sql
select * from multipage ;

SELECT * FROM page_header(get_raw_page('multipage', 0));
    lsn    | checksum | flags | lower | upper | special | pagesize |
version | prune_xid
-----------+----------+-------+-------+-------+---------+----------
+---------+-----------
 0/196C490 |        0 |     0 |    64 |  8192 |    8192 |     8192 |
4 |         0

SELECT lp, lp_flags, t_xmin, t_xmax, lp_len FROM
heap_page_items(get_raw_page('multipage', 0)) ORDER BY lp;
 lp | lp_flags | t_xmin | t_xmax | lp_len
----+----------+--------+--------+--------
  1 |        3 |        |        |      0
  2 |        3 |        |        |      0
  3 |        3 |        |        |      0
  4 |        3 |        |        |      0
  5 |        3 |        |        |      0
  6 |        3 |        |        |      0
  7 |        3 |        |        |      0
  8 |        3 |        |        |      0
  9 |        3 |        |        |      0
 10 |        3 |        |        |      0
```

The table now consists of 20 dead tuples and 10 live ones, three pages in total where the live tuples are on the last page
```sql
```

```
select n_dead_tup from pg_stat_user_tables where relname =
'multipage';
 n_dead_tup
------------
         20

select n_live_tup from pg_stat_user_tables where relname =
'multipage';
 n_live_tup
------------
         10

select pg_size_pretty(pg_relation_size('multipage'));
 pg_size_pretty
----------------
 24 kB
```

Vacuum removes all the dead rows
```sql
vacuum multipage;
select n_live_tup, n_dead_tup from pg_stat_user_tables where relname
= 'multipage';
 n_live_tup | n_dead_tup
------------+------------
         10 |          0
```
N.B. Now I have a table with no dead rows, but still 66.67% bloat!

Vacuum doesn't give space back to the system
```sql
select pg_size_pretty(pg_relation_size('multipage'));
 pg_size_pretty
----------------
 24 kB
```

Update all rows again and all live rows move to the first page again
```sql
update multipage set d = '2026-04-22';

SELECT lp, lp_flags, t_xmin, t_xmax, lp_len FROM
heap_page_items(get_raw_page('multipage', 0)) ORDER BY lp;
 lp | lp_flags | t_xmin | t_xmax | lp_len
----+----------+--------+--------+--------
  1 |        1 |    851 |      0 |    740
  2 |        1 |    851 |      0 |    740
  3 |        1 |    851 |      0 |    740
  4 |        1 |    851 |      0 |    740
  5 |        1 |    851 |      0 |    740
  6 |        1 |    851 |      0 |    740
  7 |        1 |    851 |      0 |    740
  8 |        1 |    851 |      0 |    740
```

```
  9 |        1 |    851 |      0 |    740
 10 |        1 |    851 |      0 |    740
```

At this moment the last two pages are empty and a vacuum will actually reduce the table size

```sql
vacuum multipage;
select pg_size_pretty(pg_relation_size('multipage'));
 pg_size_pretty
----------------
 8192 bytes
```

Quick Reference: pageinspect Query Cheat Sheet

```sql
-- Page header (pd_lower, pd_upper, pd_prune_xid, free space)
SELECT *, upper - lower AS free_space
FROM page_header(get_raw_page('onepage', <page_no>));

-- All line pointers + tuple headers on a page
SELECT lp AS slot, lp_flags, t_xmin, t_xmax, t_ctid, lp_off, lp_len,
  CASE lp_flags
    WHEN 0 THEN 'unused'     WHEN 1 THEN 'normal'
    WHEN 2 THEN 'redirect'   WHEN 3 THEN 'dead'
  END AS flag_desc
FROM heap_page_items(get_raw_page('onepage', <page_no>))
ORDER BY lp;

-- Infomask hint-bit decoder
SELECT lp AS slot, t_xmin, t_xmax, to_hex(t_infomask) AS
infomask_hex,
  ARRAY_REMOVE(ARRAY[
    CASE WHEN (t_infomask & x'0100'::int) != 0 THEN 'XMIN_COMMITTED'
END,
    CASE WHEN (t_infomask & x'0200'::int) != 0 THEN 'XMIN_INVALID'
END,
    CASE WHEN (t_infomask & x'0400'::int) != 0 THEN 'XMAX_COMMITTED'
END,
    CASE WHEN (t_infomask & x'0800'::int) != 0 THEN 'XMAX_INVALID'
END,
    CASE WHEN (t_infomask & x'1000'::int) != 0 THEN 'UPDATED'
END
  ], NULL) AS hint_bits
FROM heap_page_items(get_raw_page('onepage', <page_no>))
ORDER BY lp;

-- Index page items (B-tree)
SELECT itemoffset, ctid, data
FROM bt_page_items('<index_name>', <page_no>)
ORDER BY itemoffset;
```

```
-- How many live vs dead tuples (updated by VACUUM/ANALYZE)
SELECT relname, n_live_tup, n_dead_tup, last_vacuum
FROM pg_stat_user_tables WHERE relname = 'onepage';

-- Transaction status check, based on clog files
SELECT pg_xact_status('<xid>'::text::xid8);

-- Where does a row physically live?
SELECT id, ctid,
       (ctid::text::point)[0]::int AS page,
       (ctid::text::point)[1]::int AS slot
FROM onepage ORDER BY id;
```