Bloat in PostgreSQL: A Taxonomy

PostgreSQL@SCaLE20x — March 10, 2023

https://speakerdeck.com/peterg/bloat-postgresql-scale

Peter

Geoghegan

@petervgeoghegan

https://speakerdeck.com/peterg/bloat-postgresql-scale

Outline

- Most discussions of VACUUM/bloat take a practical approach

  - Starting point is VACUUM itself, and the impact ***to*** the user application

  - Can recommend “Managing Your Tuple Graveyard” talk from Chelsea Dole, which is on at 3:30 today in Ballroom A

- I’m going to take a bottom-up approach instead

  - Starting point is bloat itself, and effects that tend to naturally emerge ***from*** the user application

  - Might help you to develop a mental model that holds together existing knowledge of how these things work

  - Theoretical focus, but grounded in practicalities

https://speakerdeck.com/peterg/bloat-postgresql-scale

Overview

1. **Structure**

   Logical vs. Physical structures, TIDs as “physiological” identifiers

2. **A bottom-up take on bloat**

   Page level view of bloat, VACUUM, and opportunistic cleanup

3. **VACUUM’s priorities**

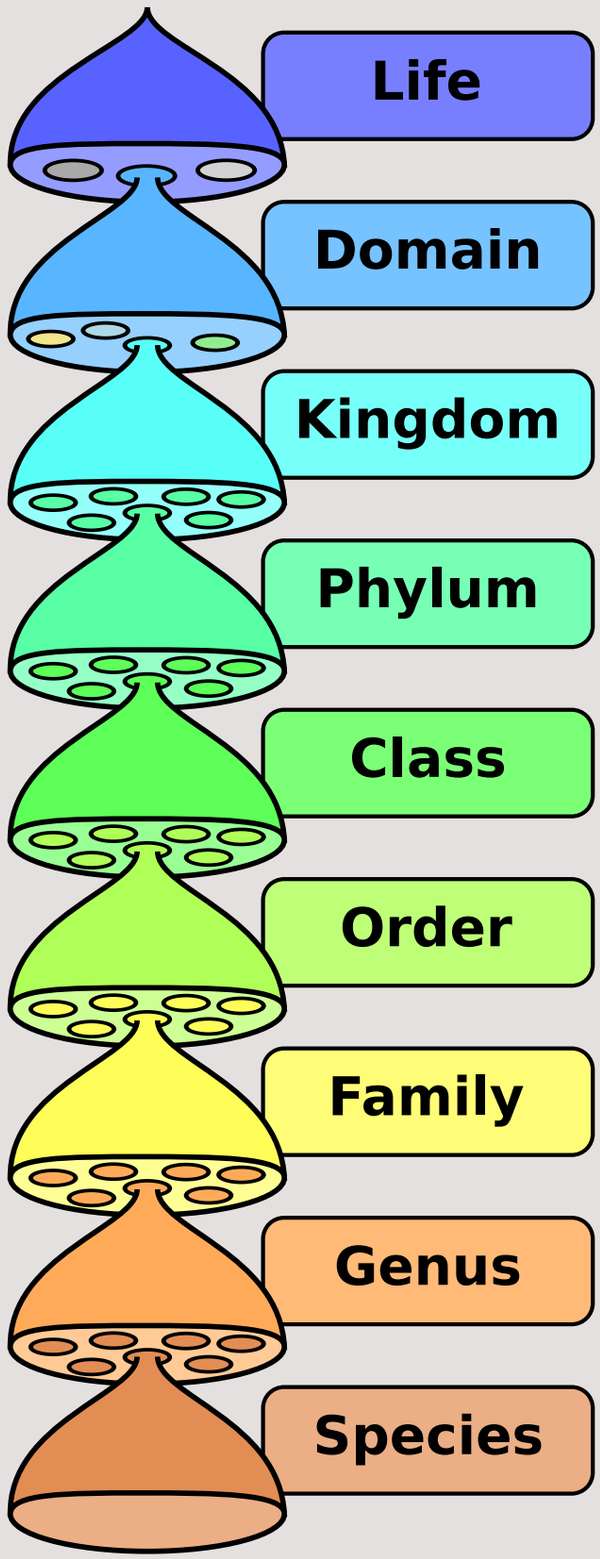
   Space reclamation, query response time

4. **Conclusions**

   Summary of the central ideas from 1 - 3

https://speakerdeck.com/peterg/bloat-postgresql-scale

Structure



Pictured: The basic scheme of modern classification (Wikipedia)

https://speakerdeck.com/peterg/bloat-postgresql-scale

Logical vs. Physical

Database systems like Postgres use access methods to abstract away physical representation.

- MVCC more or less versions entries in objects (relations).
  - “Readers don’t block writers, writers don’t block readers”
  - Baked into everything, necessitates cleaning up old versions
- Postgres heap relations (tables) generally store newly inserted tuples in whatever order is convenient.
- Index relations often have multiple versions, too.

https://speakerdeck.com/peterg/bloat-postgresql-scale

Heap (table) structure

Heap structure is optimized for sequential access, and access by index scans, which lookup entries using tuple identifiers (TIDs).

- Tuples are identified by TID (e.g., `'(2,32)'`, `'(43,89)'`), which must be *stable* so that **index scans won't break**.

- TID is a "physiological" identifier.
  - Physical *across* pages/blocks — block number.
  - Logical *within* pages/blocks — item identifier.

- "Hybrid" of logical and physical that retains many of the advantages of strictly logical and strictly physical identifiers

https://speakerdeck.com/peterg/bloat-postgresql-scale

Index structure

Indexes make access to specific records efficient through key-based searching by SQL queries.

- B-Tree indexes have strict rules about **which key values go on which "logical node"**

  - Unlike the heap, where there are no "built in" rules governing where newly inserted heap tuples can be placed

  - "Strictly logical"

- B-Tree indexes do **not** have rules about the **physical** location of any given key value

  - A page split can change the physical location of some of the entries for a given logical node

https://speakerdeck.com/peterg/bloat-postgresql-scale

Index structure (cont. 1)

Clearly the only kind of index lookup that can ever work reliably is a key search — using the whole key (or at least a prefix column)

- Going the other way (from heap entry to index entry) is harder

  - Pruning of dead heap tuples in heap pages **destroys** the information required to look up corresponding dead entries in indexes (by freeing the tuples that contain the indexed key)

  - VACUUM can only clean up indexes in bulk through a **linear scan** of each and every index, which **matches on TID** only

  - No “retail deletion” of individual entries in indexes takes place (not obvious how VACUUM could ever do this)

https://speakerdeck.com/peterg/bloat-postgresql-scale

Index structure (cont. 2)

These dependencies have important consequences for VACUUM

- They make VACUUM an inherently bulk operation, that *must* work at the level of the whole table and its indexes collectively

- Postgres uses opportunistic cleanup techniques to make up for this

  - These work at the level of individual pages, incrementally and on-demand, during query execution

  - Complements VACUUM — makes up for its weaknesses, and vice-versa

https://speakerdeck.com/peterg/bloat-postgresql-scale

Overview

1. **Structure**

   Logical vs. Physical structures, TIDs as “physiological” identifiers

2. **A bottom-up take on bloat**

   Page level view of bloat, VACUUM, and opportunistic cleanup

3. **VACUUM’s priorities**

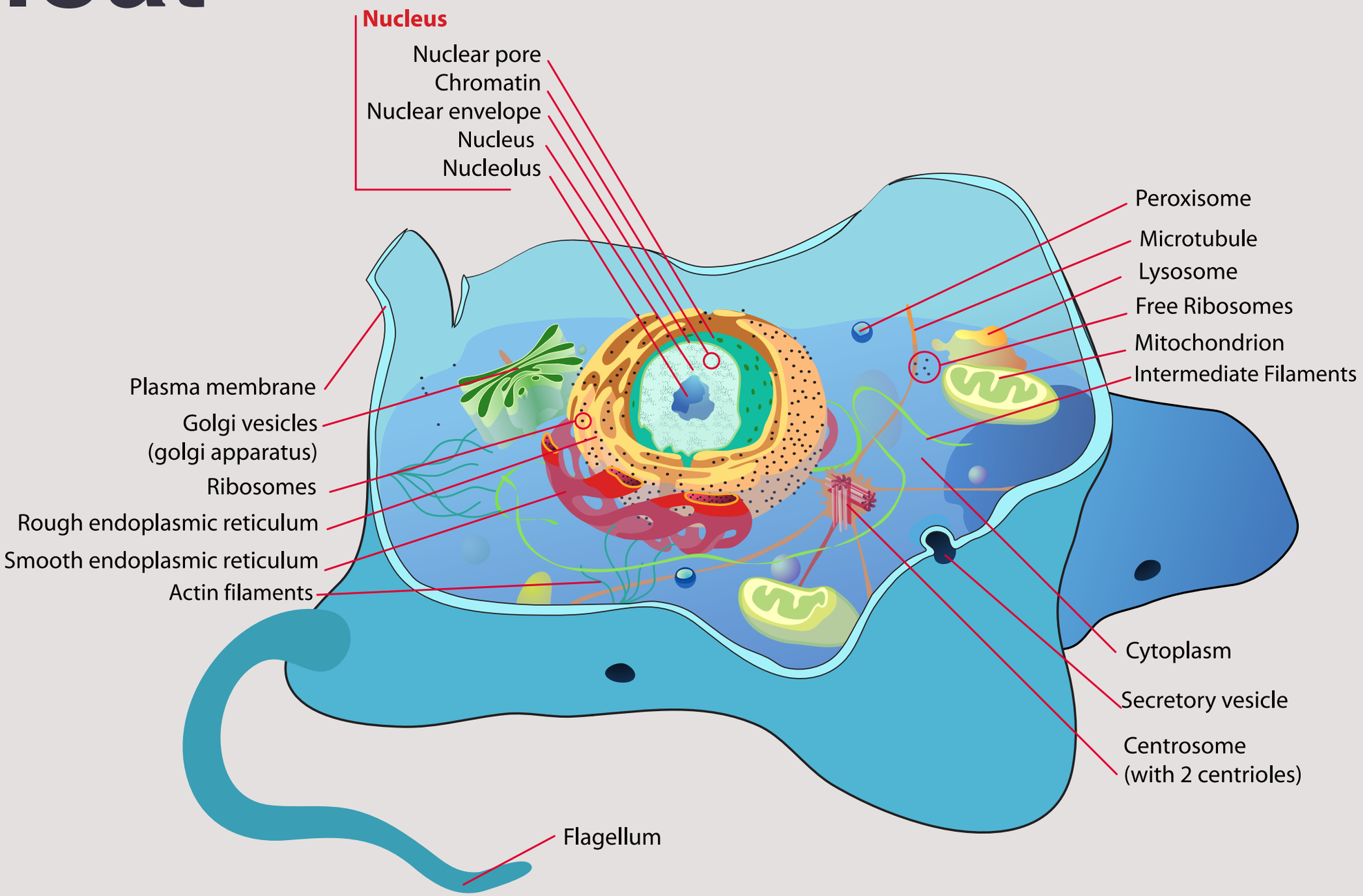
   Space reclamation, query response time

4. **Conclusions**

   Summary of the central ideas from 1 - 3

https://speakerdeck.com/peterg/bloat-postgresql-scale

A bottom-up take on bloat



Pictured: Animal Cell (Wikipedia)

https://speakerdeck.com/peterg/bloat-postgresql-scale

Database pages as “cells”

- PostgreSQL storage consists of 8KiB pages
- “Page model”
  - Individual page modifications can be made atomic with low-level techniques
  - High level atomic operations (transactions) can be composed from simpler atomic operations (WAL-logged atomic page operations)

https://speakerdeck.com/peterg/bloat-postgresql-scale

“Linux is evolution, not intelligent design”

“Bad programmers worry about the code. Good programmers worry about data structures and their relationships.”

— **Linus Torvalds**

https://speakerdeck.com/peterg/bloat-postgresql-scale

Evolutionary pressure

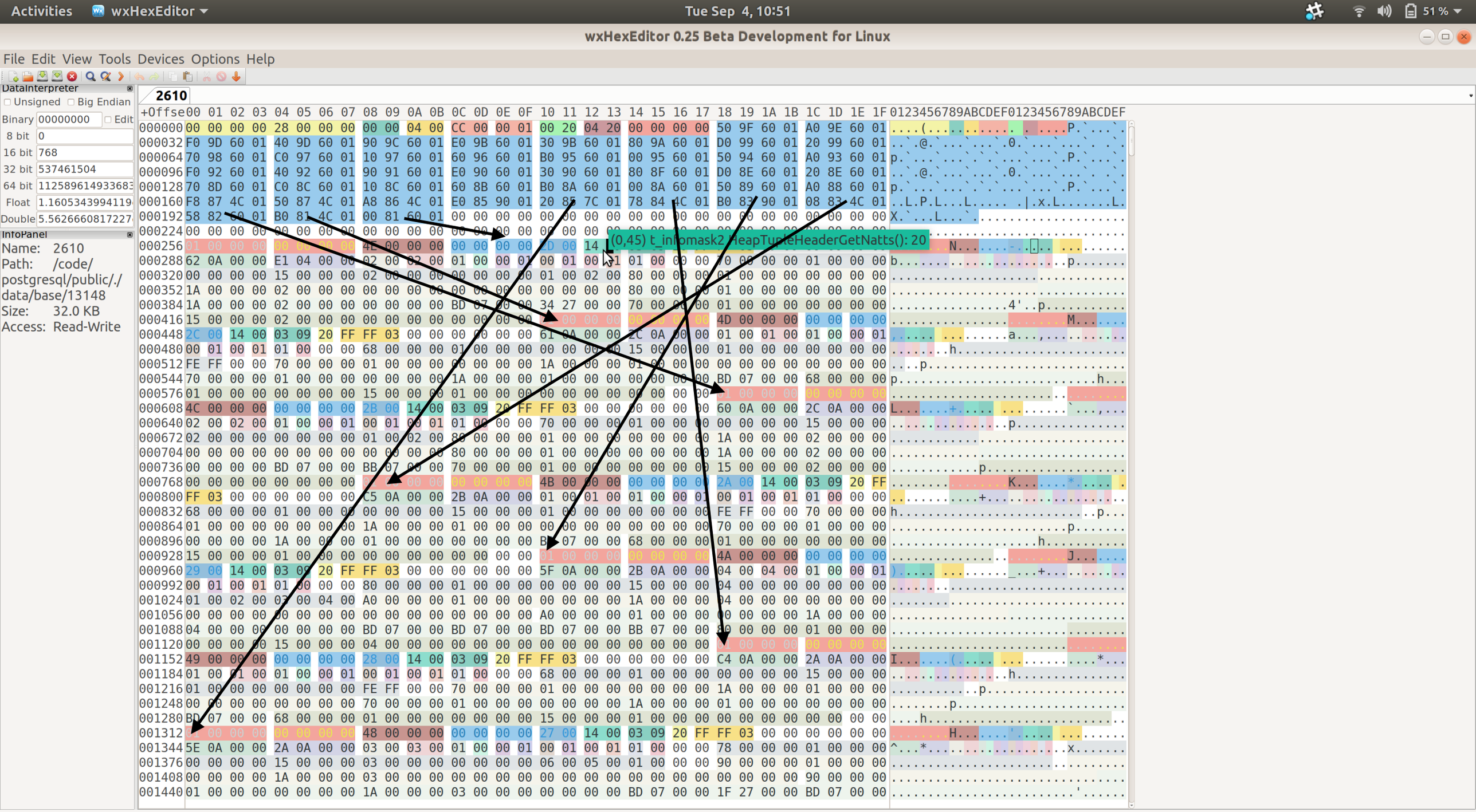
- Like cells in living organisms, the structure of pages shares a lot across disparate access methods (e.g., heap, B-Tree, …)

- The high level structures *are* very dissimilar, but the structure of individual pages is nevertheless much more similar than different

- Too much complexity to manage without breaking down into manageable pieces with commonality

  - What else could possibly work?

https://speakerdeck.com/peterg/bloat-postgresql-scale



pg_hexedit tool, with pg_index catalog relation (table)

https://speakerdeck.com/peterg/bloat-postgresql-scale

Activities wxHexEditor Tue Sep 4, 11:28 38 %

wxHexEditor 0.25 Beta Development for Linux

File Edit View Tools Devices Options Help

DataInterpreter
Unsigned Big Endian
Binary 00000101 Edit
8 bit 5
16 bit 5
32 bit 1048581
64 bit 71863393845253
Float 1.4693749450202
Double 3.5505234092499

InfoPanel
Name: 2679
Path: /code/postgresql/public/./data/base/13148
Size: 16.0 KB
Access: Read-Write

2679

```
+Offse 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 0123456789ABCDEF0123456789ABCDEF
009408 40 8D 20 00 30 8D 20 00 20 8D 20 00 10 8D 20 00 00 8D 20 00 F0 8C 20 00 E0 8C 20 00 D0 8C 20 00 @. .0. . . ... ... ... ... ... .
009440 C0 8C 20 00 B0 8C 20 00 A0 8C 20 00 90 8C 20 00 80 8C 20 00 70 8C 20 00 60 8C 20 00 50 8C 20 00 .. ... ... ... ... ... .p. .`. .P. .
009472 40 8C 20 00 30 8C 20 00 20 8C 20 00 10 8C 20 00 00 8C 20 00 F0 8B 20 00 E0 8B 20 00 D0 8B 20 00 @. .0. . . ... ... ... ... ... .
009504 C0 8B 20 00 B0 8B 20 00 A0 8B 20 00 90 8B 20 00 80 8B 20 00 70 8B 20 00 60 8B 20 00 50 8B 20 00 .. ... ... ... ... ... .p. .`. .P. .
009536 40 8B 20 00 30 8B 20 00 20 8B 20 00 10 8B 20 00 00 8B 20 00 F0 8A 20 00 E0 8A 20 00 D0 8A 20 00 @. .0. . . ... ... ... ... ... .
009568 C0 8A 20 00 B0 8A 20 00 A0 8A 20 00 90 8A 20 00 80 8A 20 00 70 8A 20 00 60 8A 20 00 50 8A 20 00 .. ... ... ... ... ... .p. .`. .P. .
009600 40 8A 20 00 30 8A 20 00 20 8A 20 00 10 8A 20 00 00 8A 20 00 F0 89 20 00 E0 89 20 00 D0 89 20 00 @. .0. . . ... ... ... ... ... .
009632 C0 89 20 00 B0 89 20 00 A0 89 20 00 90 89 20 00 80 89 20 00 70 89 20 00 60 89 20 00 50 89 20 00 .. ... ... ... ... ... .p. .`. .P. .
009664 40 89 20 00 30 89 20 00 20 89 20 00 10 89 20 00 00 89 20 00 F0 88 20 00 E0 88 20 00 D0 88 20 00 @. .0. . . ... ... ... ... ... .
009696 C0 88 20 00 B0 88 20 00 A0 88 20 00 90 88 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .. ... ... ... .................
009728 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
009760 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
009792 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
009824 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
009856 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
009888 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
009920 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
009952 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
009984 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
010016 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
010048 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
010080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
010112 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
010144 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
010176 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
010208 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
010240 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
010272 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
010304 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
010336 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................................
010368 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 08 00 0[ 00 10 00 5[ 41 00 00 00 00 00 00 ..............\A..... (1,374) indexrelid
010400 00 00 08 00 04 00 10 00 5B 41 00 00 00 00 00 00 00 00 08 00 03 00 10 00 5A 41 00 00 00 00 00 00 ........[A..........ZA.....
010432 00 00 08 00 02 00 10 00 59 41 00 00 00 00 00 00 00 00 08 00 01 00 10 00 58 41 00 00 00 00 00 00 ........YA..........XA.....
010464 00 00 07 00 2F 00 10 00 57 41 00 00 00 00 00 00 00 00 07 00 2E 00 10 00 56 41 00 00 00 00 00 00 ..../...WA..........VA.....
010496 00 00 07 00 2D 00 10 00 55 41 00 00 00 00 00 00 00 00 07 00 2C 00 10 00 54 41 00 00 00 00 00 00 ....-...UA.........,...TA.....
010528 00 00 07 00 2B 00 10 00 53 41 00 00 00 00 00 00 00 00 07 00 2A 00 10 00 52 41 00 00 00 00 00 00 ....+...SA.........*...RA.....
010560 00 00 07 00 29 00 10 00 51 41 00 00 00 00 00 00 00 00 07 00 28 00 10 00 50 41 00 00 00 00 00 00 ....)...QA.........(...PA.....
010592 00 00 07 00 27 00 10 00 4F 41 00 00 00 00 00 00 00 00 07 00 26 00 10 00 4E 41 00 00 00 00 00 00 ....'...OA.........&...NA.....
010624 00 00 07 00 25 00 10 00 4D 41 00 00 00 00 00 00 00 00 07 00 24 00 10 00 4C 41 00 00 00 00 00 00 ....%...MA.........$...LA.....
010656 00 00 07 00 23 00 10 00 4B 41 00 00 00 00 00 00 00 00 07 00 22 00 10 00 4A 41 00 00 00 00 00 00 ....#...KA.........".....JA.....
010688 00 00 07 00 21 00 10 00 49 41 00 00 00 00 00 00 00 00 07 00 20 00 10 00 48 41 00 00 00 00 00 00 ....!...IA.........  ...HA.....
010720 00 00 07 00 1F 00 10 00 47 41 00 00 00 00 00 00 00 00 07 00 1E 00 10 00 46 41 00 00 00 00 00 00 ........GA..........FA.....
010752 00 00 07 00 1D 00 10 00 45 41 00 00 00 00 00 00 00 00 07 00 1C 00 10 00 44 41 00 00 00 00 00 00 ........EA..........DA.....
010784 00 00 07 00 1B 00 10 00 43 41 00 00 00 00 00 00 00 00 07 00 1A 00 10 00 42 41 00 00 00 00 00 00 ........CA..........BA.....
010816 00 00 07 00 19 00 10 00 41 41 00 00 00 00 00 00 00 00 07 00 18 00 10 00 40 41 00 00 00 00 00 00 ........AA..........@A.....
010848 00 00 07 00 17 00 10 00 3F 41 00 00 00 00 00 00 00 00 07 00 16 00 10 00 3E 41 00 00 00 00 00 00 ........?A..........>A.....
```

Showing Page: 6 Cursor Offset: 10388 Cursor Value: 5 Selected Block: N/A Block Size: N/A

pg_hexedit tool, with pg_index_indexrelid_index catalog relation (index)

https://speakerdeck.com/peterg/bloat-postgresql-scale

Bloat at the page level

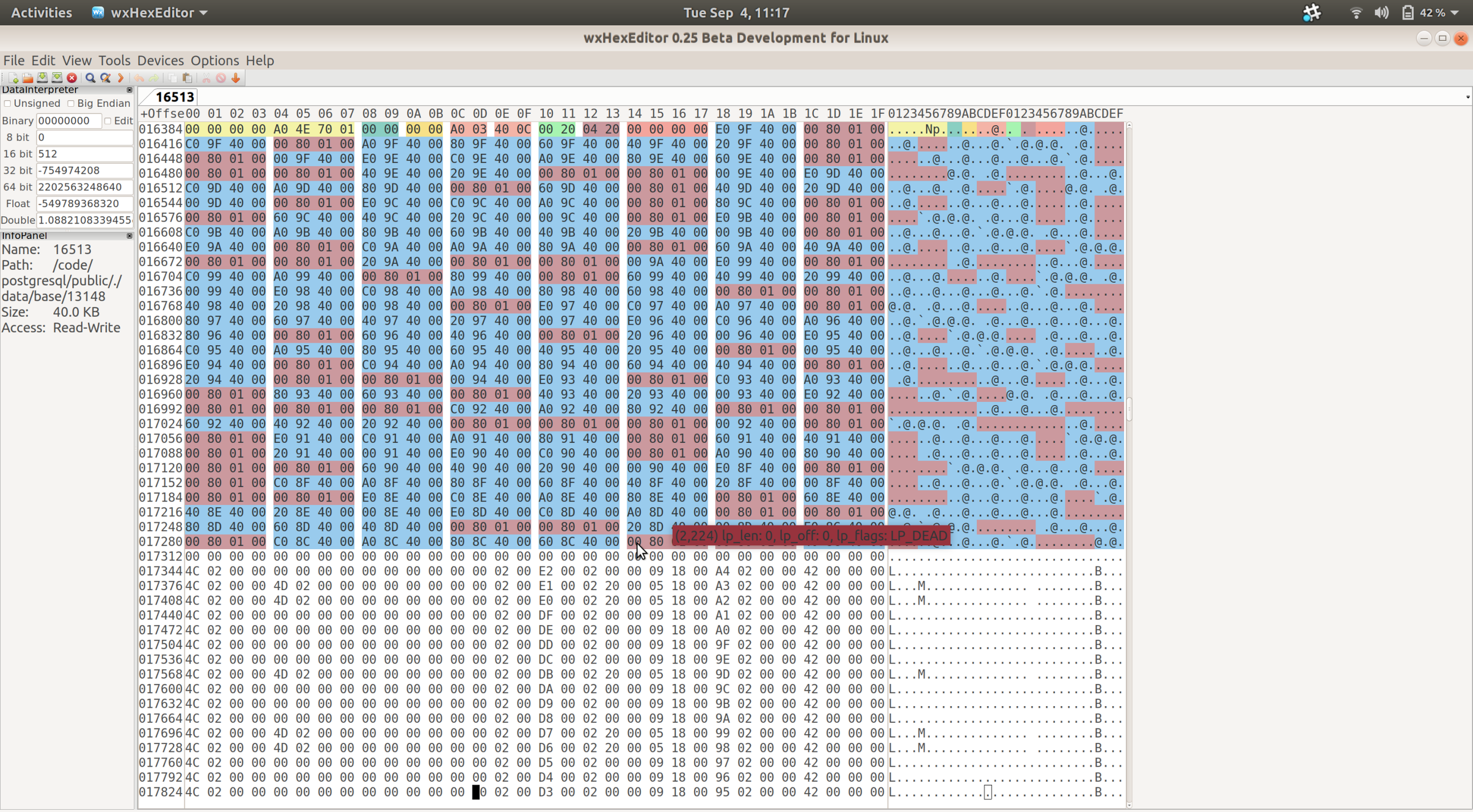
- Bloat at the level of individual pages looks similar across index and heap pages

  - Opportunistic cleanup techniques are fairly similar across heap and index pages, despite the differences that exist at the highest level (the level of whole tables)

- This is not the view that “drives” VACUUM, though

  - VACUUM is an operation that works at the level of a whole table (including its indexes) — so the “high level view” is more relevant

https://speakerdeck.com/peterg/bloat-postgresql-scale



pg_hexedit tool, with 4 byte stub dead item identifiers left behind by heap pruning

https://speakerdeck.com/peterg/bloat-postgresql-scale

VACUUM

- "Top-down" structure

  - Autovacuum (which is how VACUUM is typically run) is typically triggered by table-level threshold having been exceeded.

  - VACUUM is good at "putting a floor under" the problem of bloat at the table/system level by making sure that every table gets a "clean sweep" at some point

  - But VACUUM has **no direct understanding** of how bloat can become **concentrated** in **individual** pages, impacting queries disproportionately

- VACUUM generally processes each page once (sometimes twice), based on fixed rules

  - VACUUM from recent Postgres versions can **bypass index vacuuming** when it turns out that there are only very few entries to delete from indexes

  - But VACUUM *cannot* reorder work at runtime, nor can it add extra work at runtime

https://speakerdeck.com/peterg/bloat-postgresql-scale

VACUUM - processing order

1. Visits heap, performing pruning, collecting dead TID references needed by step 2.

    - Mechanically similar to opportunistic pruning, but directed at all pages that *might* need to be pruned, including pages that have very little bloat.

    - Pruning item identifiers as dead/LP_DEAD, since index scans still need these until after step 2 (as “tombstones”). These are the TIDs to be removed in indexes later.

2. Visits indexes, deleting index tuples that match TID list collected in step 1.

3. Second pass over heap, to mark dead/LP_DEAD item identifiers reusable/ LP_UNUSED.

    - Step 3 is more like step 2 than step 3 — same dead items TID array indicates *which* TIDs are safe to make reusable

    - Dead item TIDs (collected in step 1 and reliably removed from indexes in step 2) can now finally be marked reusable

https://speakerdeck.com/peterg/bloat-postgresql-scale

VACUUM - processing order (cont.)

Processing order makes sense when you think about basic rules around lookups

- Indexes resolve key values in heap using heap TIDs, which must be stable over time (within a VACUUM cycle).
- Index scans must *always* land on the correct heap tuple — at the very least there needs to be a “stub” 4 byte dead/LP_DEAD item identifier that serves as a tombstone to avoid *total chaos*.
  - Cannot allow index tuples/TIDs to point to who-knows-what by allowing premature recycling of TID/item identifier
- In other words, VACUUM steps 1, 2, and 3 *need* to happen in a fixed order

https://speakerdeck.com/peterg/bloat-postgresql-scale

VACUUM without steps 2 and 3

```
postgres=# create table deltable (delcol int primary key);
CREATE TABLE
postgres=# insert into deltable select generate_series(1, 1000);
INSERT 0 1000
postgres=# delete from deltable where delcol % 5 = 0;
DELETE 200
```

```
postgres=# vacuum (index_cleanup off, verbose) deltable;
INFO:  vacuuming "postgres.public.deltable"
INFO:  finished vacuuming "postgres.public.deltable": index scans: 0
pages: 0 removed, 5 remain, 5 scanned (100.00% of total)
tuples: 200 removed, 800 remain, 0 are dead but not yet removable
removable cutoff: 275362920, which was 0 XIDs old when operation ended
new relfrozenxid: 275362918, which is 1 XIDs ahead of previous value
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan bypassed: 5 pages from table (100.00% of total) have 200 dead item identifiers
avg read rate: 134.698 MB/s, avg write rate: 202.047 MB/s
buffer usage: 14 hits, 2 misses, 3 dirtied
WAL usage: 6 records, 0 full page images, 858 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

https://speakerdeck.com/peterg/bloat-postgresql-scale

Finishing off the steps we skipped (steps 2 and 3)

```
postgres=# vacuum (index_cleanup on, verbose) deltable;
INFO:  vacuuming "postgres.public.deltable"
INFO:  finished vacuuming "postgres.public.deltable": index scans: 1
pages: 0 removed, 5 remain, 5 scanned (100.00% of total)
tuples: 0 removed, 800 remain, 0 are dead but not yet removable
removable cutoff: 275362920, which was 0 XIDs old when operation ended
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan needed: 5 pages from table (100.00% of total) had 200 dead item identifiers removed
index "deltable_pkey": pages: 5 in total, 0 newly deleted, 0 currently deleted, 0 reusable
avg read rate: 0.000 MB/s, avg write rate: 70.383 MB/s
buffer usage: 31 hits, 0 misses, 1 dirtied
WAL usage: 15 records, 1 full page images, 10058 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

https://speakerdeck.com/peterg/bloat-postgresql-scale

HOT UPDATEs

Heap-only tuple optimization avoids creating new index entries during updates

- Chains together heap tuples in heap page — existing TIDs in indexes can find the version of interesting by traversing the HOT chain (version chain) from the heap page
- Avoids adding successor versions to indexes *in the first place* (hence the name “heap-only tuple”).
- Optimization only applies when **no** indexed columns are modified by UPDATE statements.
  - “All or nothing” — successor versions in indexes either avoided entirely, or required in each and every index

https://speakerdeck.com/peterg/bloat-postgresql-scale

Opportunistic ~~HOT~~ pruning

- "Prunes away" heap tuples during query execution (not during VACUUM), which is important for many workloads

  - Including workloads that don't manage any HOT updates! "HOT pruning" is a historical **misnomer** that caught on — *all* heap tuples can be pruned

- Cannot reclaim dead item identifiers in the heap page, except with heap-only tuples

  - Heap-only tuples are not *directly* referenced from indexes/TIDs (only indirectly), so there is no "3 steps of VACUUM" style dependency to worry about when pruning HOT chains

  - Pruning *will* free a little more space in affected heap pages when there is only ever "heap-only tuple bloat" (left by HOT updates)

  - But this may not be very significant **at the level of the heap page itself** (indexes are another matter). Far more space used for heap tuples than dead stub item identifiers, which take up only 4 bytes — **far less** than the tens or even hundreds of bytes it takes to store **tuples themselves**.

https://speakerdeck.com/peterg/bloat-postgresql-scale

Opportunistic index deletion

- B-Tree also independently cleans up bloat opportunistically, at the level of individual pages

- Postgres 14 greatly improved this mechanism, by making it specifically target non-HOT update bloat in indexes

  - Limits build-up of bloat in individual index pages

  - “All or nothing” nature of HOT update optimization *is* still a problem — but the worst case is vastly improved

  - Can perfectly preserve the size of indexes affected by many non-HOT updates

  - Index deduplication (added in Postgres 13) also helps by “soaking up” bursts of duplicates needed for versioning purposes

https://speakerdeck.com/peterg/bloat-postgresql-scale

“Top-down” VACUUM versus “bottom-up” opportunistic cleanup

- VACUUM works at the level of a **whole table and its indexes**, collectively

  - Top-down, global

  - “Puts a floor under” level of bloat in table and indexes collectively

- Opportunistic techniques work at the level of **individual pages**

  - Bottom-up, local

  - Limits the **concentration** of bloat in individual pages

  - Runs during query processing, as often as required

  - “Holds the line” for VACUUM, since VACUUM doesn’t “understand” the ways in which different pages (from the same table/index) sometimes have dramatically different requirements

https://speakerdeck.com/peterg/bloat-postgresql-scale

“Deleting a million rows once” versus “deleting one row a million times”

- There is not too much difference…in theory
- The practical differences are far greater than you might guess
- A “problem within a table” versus a “problem within a page”
  - VACUUM/autovacuum is typically much more effective at cleaning up after a bulk delete
  - Opportunistic techniques require…opportunities! In general, there may not be any SQL queries that try to read deleted data.
  - Opportunistic techniques enable reuse of space for **new versions** of **nearby, related** rows — which avoids fragmentation

https://speakerdeck.com/peterg/bloat-postgresql-scale

Overview

1. **Structure**

   Logical vs. Physical structures, TIDs as “physiological” identifiers

2. **A bottom-up take on bloat**

   Page level view of bloat, VACUUM, and opportunistic cleanup

3. **VACUUM’s priorities**

   Space reclamation, query response time

4. **Conclusions**

   Summary of the central ideas from 1 - 3

VACUUM's Priorities

VACUUM design goals

VACUUM is designed to be non-disruptive.

- Heavyweight lock strength doesn't block user queries, including INSERTs, UPDATEs, and DELETEs.

- Indexes are scanned in physical order during VACUUM, not logical order (B-Tree and GiST only).

- Preserving worst case query response time is arguably the primary goal.

  - Not impacting response time *while VACUUM runs* matters almost as much

  - Reclaiming space is only a secondary goal.

https://speakerdeck.com/peterg/bloat-postgresql-scale

Space reclamation

Space reclamation *is* important in extreme cases

- Modest amounts of free space can be reclaimed eventually in more common cases, where query response time matters most.

- VACUUM *occasionally* truncates heap tables, giving space back to the operating system

- Indexes have their own unique restrictions on space reuse

  - Only whole index pages can be reclaimed by the free space map — undersized pages cannot be merged together.

  - Look out for “pages deleted” for indexes in autovacuum log output (on Postgres 14+)

https://speakerdeck.com/peterg/bloat-postgresql-scale

VACUUM and table size

Big tables vs. small tables

- Big tables (however you define them) aren't processed any differently than small tables by VACUUM

  - But they should be thought of as *qualitatively* different things in practice

- Bigger tables *must* receive fewer individual VACUUM operations, each of which will be longer and more expensive (compared to a similar table with far fewer rows)

  - The table doesn't stop accumulating garbage while VACUUM runs

  - But VACUUM only removes tuples that were **already** garbage **when the VACUUM operation began**

  - **"Too big to fail"** dynamics may come into play (e.g., autovacuum cancellation can hurt a lot more with larger tables)

https://speakerdeck.com/peterg/bloat-postgresql-scale

VACUUM and table size (cont.)

Breaking big VACUUMs down into smaller VACUUMs is a good strategy

- Table partitioning can help with this

  - Individual partitions/child tables are processed as independent tables by VACUUM

- More frequent VACUUMs by autovacuum *may* also help

  - Works best with append-mostly tables with few or no garbage tuples

  - Number of pages scanned by each VACUUM (as opposed to skipped using the visibility map) is important — heap pages set all-visible must remain all-visible rather than being processed/scanned again and again.

  - Postgres 15 was the first version that instrumented “pages scanned” in autovacuum log reports (as well as in VACUUM VERBOSE)

https://speakerdeck.com/peterg/bloat-postgresql-scale

Incremental autovacuum of a large and continually growing table, with updates

```
automatic vacuum of table "postgres.public.order_line": index scans: 0
pages: 0 removed, 7385017 remain, 2091215 scanned (28.32% of total)
tuples: 4964150 removed, 451619205 remain, 2728993 are dead but not yet removable
removable cutoff: 76733064, which was 2311556 XIDs old when operation ended
frozen: 406579 pages from table (5.51% of total) had 24215117 tuples frozen
index scan bypassed: 124263 pages from table (1.68% of total) have 406397 dead item identifiers
avg read rate: 28.617 MB/s, avg write rate: 19.705 MB/s
buffer usage: 2115525 hits, 1945133 misses, 1339404 dirtied
WAL usage: 2042719 records, 405011 full page images, 2881770710 bytes
system usage: CPU: user: 14.81 s, system: 18.83 s, elapsed: 531.03 s
```

https://speakerdeck.com/peterg/bloat-postgresql-scale

Overview

1. **Structure**

   Logical vs. Physical structures, TIDs as “physiological” identifiers

2. **A bottom-up take on bloat**

   Page level view of bloat, VACUUM, and opportunistic cleanup

3. **VACUUM’s priorities**

   Space reclamation, query response time

4. **Conclusions**

   Summary of the central ideas from 1 - 3

Conclusions about bloat in PostgreSQL

- VACUUM is tasked with removing old garbage tuples that are obsolete to all possible transactions.

  - Recent Postgres versions are much better at showing you what's really going on

- Opportunistic techniques (HOT Pruning, deletion in B-Tree indexes) also exist. Garbage collection usually happens both ways.

  - Even recent Postgres versions make it hard to tell how much of this has happened

  - But can be inferred from VACUUM instrumentation (particularly autovacuum logging), to a degree

https://speakerdeck.com/peterg/bloat-postgresql-scale

Conclusions about bloat in PostgreSQL (cont.)

- Opportunistic techniques are restricted by the same ordering requirements that dictate high-level steps VACUUM performs.

  - Index scans cannot be allowed to land on an unrelated tuple due to heap TID recycling.

  - So even workloads/tables that do a great deal of cleanup opportunistically are bound to eventually require vacuuming to mark dead item identifiers for reuse

- There is an important complementary relationship between VACUUM and opportunistic cleanup

https://speakerdeck.com/peterg/bloat-postgresql-scale