Bloat in PostgreSQL: A Taxonomy

PostgreSQL@SCaLE20x – March 10, 2023

Peter Geoghegan @petervgeoghegan

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



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

Structure



Pictured: The basic scheme of modern classification (Wikipedia)

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.

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

Index structure

Indexes make access to specific records efficient through keybased 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

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)

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



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



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)

"Linux is evolution, not intelligent design"

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

— Linus Torvalds

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?

Block Size: N/A

 $-\Box$

wxHexEditor	0.25 B	eta Deve	lopment	for Linux
-------------	--------	----------	---------	-----------

File	Edit	View	Tools	Devices	Options	Help	
------	------	------	-------	---------	---------	------	--

DataInterpreter 🛛 🛛		0 0	/ 🌵																									
□ Unsigned □ Big Endian	2610																											
Binary 0000000 C Edit	+Offse0	0 01	02	03 (94 0	5 06	07	08 0	9 0A	0B	0C (9D 0	0E 01	F 10	11	12 :	13 1	4 15	16	17	18	19 1	A 18	10	1D :	1E 1	1F (0123456789ABCDEF0123456789ABCDEF
	0000000	00 0	00	00 2	28 0	0 00	00	00 0	0 04	00	CC (00	0 01	1 00	20	04 2	20 0	0 00	00	00	50	9F 6	0 01	AO	9E (60 0	91	(
	000032 F	0 9D	60	01 4	40 91	D 60	01	90 9	C 60	01	E0 9	9B 6	60 OI	1 30	9B	60 0	91 8	0 9A	60	01	D0 9	99 6	0 01	20	99 (60 0	91	`.@.```.0.````.
16 bit 768	0000647	98 0	60	01 (CO 9	7 60	01	10 9	7 60	01	60 9	96 6	60 01	1 B0	95	60 0	91 0	0 95	60	01	50	94 6	0 01	A 0	93 (60 0	91 J	p.```.`.``.P.``.
32 bit 537461504	000096 F	0 92	60	01 4	40 93	2 60	01	90 9	1 60	01	E0 9	90 6	60 01	1 30	90	60 0	91 8	0 8F	60	01	D0 8	8E 6	0 01	20	8E (60 0	91	`.@.```.0.```.
64 bit 112589614933683	0001287	0 8D	60	01 (0 8	C 60	01	10 8	C 60	01	60 8	BB 6	60 01	1 B0	8 A	60 0	91 0	0 8A	60	01	50	89 6	0 01	A0	88 (60 0	91 J	o.``.`.`.```.P.``.
Float 1.1605343994119	000160 F	8 87	4C	01 5	50 8	7 4C	01	A8 8	6 4C	01	E0 8	35 9	0 01	1 20	85	7C (91 7	8 84	4C	01	B0 8	83/9	0 01	08	83	4C (91	L.P.LL .x.LL.
Double 5 5626660817227	000192 5	8 82	60	01 E	30 8	1.40	01	00 8	1 60	01	00 0	00 0	0 00	9 00	00	00 0	90 0	0 00	00	00	00	0 0	0 00	00	00 (00 0	00	Χ.`L`.
IntoPanel	0002240	0 00	00	00 (0 00	0 00	00	00 0	0 00	00	00 (00.0	0 00	00	00	00 (0	0.00	- 10	<u> </u>	00		0 00	00		00 (
Name: 2610	000256	1 00	00	00	0.0	0.0	00	4E 0	0 00	00	00 0	00 0	0 00		00	14.	(0,4	5) t_i	infor	nask	(2) Ae	eapT	uptel	lead	lerGe	etNa	atts	s(): 20 N
Dath: /code/	0002886	2 0A	00	00 E	E1 04	4 00	00	02 0	0 02	00	01 0	00 0	0 01	00	01	00	€1 0	1 00	00	00	10	00 0	0 00	01	00 (00 0	90 H	D
	0003200	0 0 0	00	00 1	15 0	0 00	00	02 0	0.00	00	00 (00 0	0	01	00	02 0	00 8	0 00	00	00	07	00 0	0 00	00	00 (00 0	90	
postgresqt/public/./	0003521	A 00	00	00 0	92 0	0 00	00	00 0	0 00	00	00		00 00	9 00	00	00 0	00 8	0 00	00	00	01	00 0	0 00	00	00 (00 0	90	
data/base/13148	000384 1	Δ 00	00	00 0	92 0	0 00	00	00 0	0 00	00	RD (7 0	0 00	9 34	27	00 0	90 7	0 00	D.	700	01	00 0	0 00	00	00 (20	4' n
Size: 32.0 KB	0004161	5 00	00	00 0	92 0	0 00	00	00 0	0 00	0.0	00 0		0 00		00	00 0			~ X	0.0	4D (00 0	0 00	0.0	00		30	М
Access: Read-Write	0004482		14	00 0	92 00	9 20	FF	EE 0	3 00	0.0	00 0	h		0.61	-0.0	00 0	<u>.</u>	C 04		00	01	00 0	1 00	01	00	00 0	31	а
	0004402	0 01	00	01 0	01 0		00	68 0	00 00	00	01/0			0 0 0	0.0	00 0	10 1	5 00	00	00	01	00 0	0 00	00	00 0		30	h
	000512 E		00	00 7		0 00	00	00 0	0000	00				ο 1Λ	00	00 0	00 0		00	00	001	00 0		00	00 0		30	n
	0005121		00	00 0	ຸດ 0,	0000	00	00 0	0000	00				0 01.	00		0 0		00	00	BD (000	0 00	68	00 0		30	h
	0005760	1 00	00			0000	00	15 0	0 00		01 0				00		0 00		00-	00				00	00 (00 (50 J	
	0005700		00				00	15 0 2P 0			03 0				00				00	00	60			20	ΩΛ (00 0	ວດ	
	0000004		00	00			01			00					00			1 00			00			15	00 0			·····
	0000400		02				00	00 0		00	00 0				00			0 00			1 4			02	00 0			······································
	0000720		00				00	01 0	0 02	00					00	00 0								02				••••••••••••••••
	0007040		00		0 00		00			00					00	0000				00	1A 1			02	00 0		20	
	0007600		00	00 0	30 0		00	BBU		00	70 0	90 0	0 00		00					00	12 1		4 00	02				p
			00				00							4B	00				00	01			4 00	03	09			· · · · · · · · · · · · · · · · · · ·
	000800 F	F 03	00				00		A 00	00	ZB	JA C			00				00	00	00		0 01	01	00 0		90	•••••••••••••••••••••••••••••••
	0008320	8 00	00	00 0				00 0		00	15 0			9 01	20	00 0			00	00	FE I		0 00	70	00 0			np
	0008640		00	00 0			00			00	UI (00 00	00	00 0	0000		00	00	/0 0		0 00	01	00 0		90	p
	0008960		00	00			00	01 0		00	00 0			9 B	>07	00 0	96 96	8 00	00	00	UL (0 00	00	00 0		90	n
	0009281	5 00	00	00 0			00	00 0		00	00 0				00	00 0			00	00	4A		0 00	00	00			· · · · · · · · · · · · · · · · · · ·
	0009602	9 00	14	00 0	03 0	20	FF	FF 0	3 00	00	00 0	90 6		9 5F	0A	00 0	00 2	B 0A	00	00	04	00 0	4 00	01	00 0	00 0	91)
	0009920	0 01	00	01 0		0 00	00	80 0	0 00	00	01 0	90 6	00 00	9 00	00	00 0	90 I	5 00	00	00	94 (00 0	0 00	00	00 (00 0	90	•••••••••••••••••••••••••••••••••••••••
	0010240	1 00	02	00 0	0 0	0 04	00	A0 0	0 00	00	01 0	90 6	00 00	9 00	00	00 0	90 I	A 00	00	00	94 (00 0	0 00	00	00 (00 0	90	•••••••••••••••••••••••••••••••••••••••
	0010560	00 00	00	00	00 00	0 00	00	00 0	0 00	00	00 0	90 6	00 00	9 A0	00	00 0	90 0	1 00	00	00	00	00 0	0 00	1A	00 (00 0	90	•••••••••••••••••••••••••••••••••••••••
	0010880	4 00	00	00 0	00 00	0 00	00	BD 0	/ 00	00	BD (976	00 00	9 BD	0/	00 0	90 B	B 0/	00	00	80	00 0	0 00	01	00 (00 0	90	• • • • • • • • • • • • • • • • • • • •
	0011200	0 00	00	00 1	15 0	0 00	00	04 0	0 00	00	00 0	00 0	00 00	9 00	00	00 0	0 00	0 00	00	00	61 (00 0	0 00	00	00 (00 (910	• • • • • • • • • • • • • • • • • • • •
	001152 4	9 00	00	00	0 00	0 00	00	28 0	0 14	00	03 (99 2	20 FF	F FF	03	00 0	90 0	0 00	00	00	C4 (0A 0	0 00	2A	0A (00 0	90 .	I
	001184 0	1 00	7 1	00	91 0	0 00	01	00 0	1 00	01	01 (00 0	00 00	9 68	00	00 0	90 0	1 00	00	00	00	00 0	0 00	15	00 (90 0	90	<mark></mark> h
	001216 0	1 00	00	00 0	0 00	0 00	00	FE F	F 00	00	70 0	00 0	00 00	9 01	00	00 0	90 0	0 00	00	00	1A (00 0	0 00	01	00 (00 0	90	p
	001248 0	0 0	00	00 0	0 00	0 00	00	70 0	0 00	00	01 (00 0	0 00	9 00	00	00 0	90 1	A 00	00	00	01 (00 0	0 00	00	00 (00 0	90	pp
	001280 B	0/07	00	00	58 0	0 00	00	01 0	0 00	00	00 0	00 0	0 00	9 15	00	00 0	90 0	1 00	00	00	00	00 0	0 00	00	00	90 0	90	h
	001312	1 00		00			00	48 0	0 00	00	00 (00	0 00	9 27	00	14 (0.00	3 09	20	FF	FF (03 0	0 00	00	00 (00 00	90	· · · · · · · · · · · · · · · · · · ·
	001344 5	E 0A	00	00 2	2A 0/	A 00	00	03 0	0 03	00	01 0	00 0	00 01	1 00	01	00	91 0	1 00	00	00	78 (00 0	0 00	01	00 (00 0	90 í	^*
	001376 0	00 0	00	00 1	15 0	0 00	00	03 0	0 00	00	00 0	00 0	00 00	9 06	00	05 0	0 00	1 00	00	00	90	00 0	0 00	01	00 (00 0	90	
	001408 0	00 0	00	00 1	1A 0	0 00	00	03 0	0 00	00	00 0	00 0	0 00	00 0	00	00 0	0 00	0 00	00	00	00	00 0	0 00	90	00 (00 0	90	
	001440 0	1 00	00	00 0	0 00	0 00	00	1A 0	0 00	00	03 0	00 0	0 00	9 00	00	00 0	90 B	D 07	00	00	1F :	27 0	0 00	BD	07 (00 00	90	
																												v
Showing Page: 0					C	ursor	Offset	t: 275								Cur	sor Va	alue: (0								S	elected Block: N/A

pg_hexedit tool, with pg_index catalog relation (table)

File Edit View Tools Devices Options Help

🔒 🔛 🖳 🙁 🔕 🔍 🗶 📏 🛛	(h) / h) h) h)	X 🛇 🕹 🗌																			
Unsigned Big Endian	2679																				
inary 00000101	+Offse00	01 02 0	03 04 0	5 06 0	7 08 0	9 0A 0	B 0C	0D 0E	0F	10 11	. 12 1	3 14	15 16	6 17	18	19 1A	A 1B	1C 1	LD 1E	E 1F 0123456789ABCDEF0123456789ABCDEF	
R hit 5	009408 40	8D 20 0	00 30 81	0 20 0	0 20 8	D 20 0	0 10	8D 20	00	00 8D	20 0	9 F0	8C 20	9 00	E0 8	BC 20	00	D0 8	BC 20	9 00 <mark>00</mark>	
6 bit 5	009440 C0	8C 20 0	00 B0 80	20 0	0 A0 8	C 20 0	0 90	8C 20	00	80 80	20 0	9 70	8C 20	9 00	60 8	BC 20	00	50 8	3C 20	9 00p`P	
2 bit 1049591	009472 40	8C 20 0	00 30 80	20 0	0 20 8	C 20 0	0 10	8C 20	00	00 80	20 0	9 F0	8B 20	9 00	E0 8	3B 20	00	D0 8	3B 20	9 00 @0	
4 bit 71963202945252	009504 0	8B 20 0	00 B0 81	3 20 0	0 A0 8	B 20 0	0 90	8B 20	00	80 88	20 0	9 70	88 20	9 00	60 8	3B 20		50 8	SB 20	J 00pP	
4 bit 71003393045253	00953640	88 20 C		3 20 0		B 20 0		SR 20	00		20 0		VA ZU		EU 0	3A 20			SA 20		
-loat 1.4693749450202	009508 00	0A 20 0	00 30 8/	A 20 0	0 00 0 0 20 8	A 20 0	0 90	0A 20 8A 20	00	00 00			0A 20 80 26	000		0A 20 20 20			0A 20	ооса о	
ouble 3.5505234092499	00900040	80 20 0	0 30 8/	A 20 0	0 20 0	A 20 0	0 00	89 20	00	80 80		9 70	89 20 89 20	9 00	60 8	20 20 29 20		50 8	09 20 09 20	ο οο (μ ο	
ItoPanel 🛛 🕷	00966440	89 20 6	0 30 80	200	0 20 8	9 20 0	0 10	89 20	00	00 89	20 0	9 F0	88 26	9 00	F0 8	38 20	00	00 8	38 20	9 00 a 0	
alle. 2019	009696 00	88 20 0	00 B0 88	3 20 0	0 A0 8	8 20 0	0 90	88 20	00	00 00	00 0	00	00 00	00 6	00 (00 00	00	00 0	0 00	00	
action (Coue)	00972800	00 00 0	0 00 00	0000	0 00 0	0 00 0	0 00	00 00	00	00 00	00 0	00 C	00 00	9 00	00 (00 00	00	00 0	00 00	00	
ostgresqt/public/./	00976000	00 00 0	0 00 00	0 00 0	0 00 0	0 00 0	0 00	00 00	00	00 00	00 0	9 00	00 00	9 00	00 (00 00	00	00 0	00 00	00	
ala/Dase/15148	009792 00	00 00 0	0 00 00	0 00 0	0 00 0	0 00 0	0 00	00 00	00	00 00	00 0	00	00 00	00	00 (00 00	00	00 0	00 00	00	
IZE: IO.U KB	00982400	00 00 0	0 00 00	0 00 0	0 00 0	0 00 0	0 00	00 00	00	00 00	00 0	9 00	00 00	9 OO	00 (90 00	00	00 0	00 00	9 00	
CCESS. Read-WITLE	009856 00	00 00 0	0 00 00	0 00 0	0 00 0	0 00 0	0 00	00 00	00	00 00	00 0	9 00	00 00	9 00	00 (00 00	00	00 0	00 00	9 00	
	009888 00	00 00 0	0 00 00	0 00 0	0 00 0	0 00 0	0 00	00 00	00	00 00	00 0	9 00	00 00	9 00	00 (00 00	00	00 0	00 00	9 00	
	00992000	00 00 0	0 00 00	0 00 0	0 00 0	0 00 0	0 00	00 00	00	00 00	00 0	9 00	00 00	9 00	00 (00 00	00	00 0	00 00	9 00	
	009952 00	00 00 0	0 00 00	0 00 0	0 00 0	0 00 0	0 00	00 00	00	00 00	00 0	9 00	00 00	9 00	00 (00 00	00	00 0	00 00	9 00	
	009984 00	00 00 0	0 00 00	0 00 0	0 00 0	0 00 0	0 00	00 00	00	00 00	00 0	9 00	00 00	9 00	00 (00 00	00	00 0	00 00	9 00	
	01001600	00 00 0	00 00 00	0 00 0	0 00 0	0 00 0	0 00	00 00	00	00 00	00 0	9 00	00 00	9 00	00 (90 00	00	00 0	00 00	9 00	
	01004800	00 00 0		0000	0 00 0	0 00 0	0 00	00 00	00	00 00	0000	9 00		9 00	00 0	00 00	00 0	00 0		J 00	
	01018000								00						00 0						
	01011200								00											9 00	
	010144 00				0 00 0				00			000				00 00				9 00	
	01017000			0000	0 00 0		0 00	00 00	00			000	00 00	00 0	00 0	00 00	00	00 0	00 00	9 00	
	010240 00	00 00 0	0 00 00	0000	0 00 0	0 00 0	0 00	00 00	00	00 00	00 0	9 00 9 00	00 00	9 00 9 00	00 (00 00	00	00 0	00 00	9 00	
	010272 00	00 00 0	0 00 00	0 00 0	0 00 0	0 00 0	0 00	00 00	00	00 00	00 0	9 00 C	00 00	9 00	00 (00 00	00	00 0	00 00	9 00	
	010304 00	00 00 0	0 00 00	0 00 0	0 00 0	0 00 0	0 00	00 00	00	00 00	00 0	9 00	00 00	9 00	00		00	00 0	00	00	
	010336 00	00 00 0	0 00 00	0 00 0	0 00 0	0 00 0	0 00	00 00	00	00 00	00 0	9 00	00 00	9 00	₀₀ (1	1,374)) inde	exreli	id ₀₀	9 00	
	010368 00	00 00 0	0 00 00	0 00 0	0 00 0	0 00 0	0 00	00 00	00	00 00	08 0	0	00 10	9 00	56.4	41 00	00	00 0	00 00	9 00	
	01040000	00 08 0	00 04 00	10 0	<mark>0</mark> 5B 4	1 00 0	0 00	00 00	00	00 00	08 0	9 0 3	00 <mark>10</mark>	<u>9 00</u>	5Å	41 00	00	00 0	00 00	00 [A	
	010432 00	00 08 0	00 02 00	10 0	<mark>0</mark> 59 4	1 00 0	0 00	00 00	00	00 00	08 0	9 01	00 <mark>10</mark>	<u>9 00</u>	58 4	41 00	00	00 0	00 00	00YAXA	
	010464 00	00 07 0	00 2F 00	10 0	<mark>0</mark> 57 4	1 00 0	0 00	00 00	00	00 00	07 0	9 2E	00 <mark>10</mark>	9 00	56 4	41 00	00	00 0	00 00	9 00/. <mark></mark> WAVA	
	010496 00	00 07 0	00 2D 00	10 0	0 55 4	1 00 0	0 00	00 00	00	00 00	07 0	9 2C	00 10	9 00	54 4	41 00	00	00 0	00 00	9 00UA,	
	01052800	00 07 0	00 2B 00	10 0	0 53 4	1 00 0	0 00	00 00	00	00 00	07 0	9 2A	00 10	9 00	52 4	41 00	00	00 0	00 00	9 00+SA**	
	01056000	00 07 0	00 29 00	10 0	0 51 4	1 00 0	0 00	00 00	00	00 00	07 0	9 28	00 10	9 00	50 4	41 00	00	00 0	00 00	9 00)QA(PA	
	01059200	00 07 0	$\frac{10}{27}$ $\frac{27}{27}$ $\frac{00}{27}$			1 00 0	0 00	00 00	00		070	26		9 00	4E 4	41 00	00	00 0	00 00	J 00	
	01062400					1 00 0			00			9 24 D 22			40 4	41 00 41 00				J UU	
	01065800					1 00 0			00		070	9 ZZ			4A 4	11 00				σο τ το μο	
	010720 00				0 49 4	1 00 0	0 00	00 00	00		07 0	9 1E		9 00	40 4	11 00		00 0		ο θθ	
	01075200	00 07 0			0 45 4	1 00 0	0 00	00 00	00	00 00	07 0	$\frac{1}{2}$	00 10	9 00	44 4	11 00	00	00 0	00 00	9 00 ΕΑ	
	010784 00	00 07 0	10 10 00		0 43 4	1 00 0	0 00	00 00	00	00 00	07 0	9 1A	00 10	00 0	42 4	41 00	00	00 0	0 00	00 CA	
	010816 00	00 07 0	00 19 00	10 0	0 41 4	1 00 0	0 00	00 00	00	00 00	07 0	9 18	00 10	9 00	40 4	11 00	00	00 0	00 00	AA	
	01084800	00 07 0	00 17 00	10 0	0 3F 4	1 00 0	0 00	00 00	00	00 00	07 0	9 16	00 10	00 0	3E 4	41 00	00	00 0	00 00	9 00	
howing Page: 6			C	irsor Off	set: 1038	88					Curs	or Valu	e 5							Selected Block: N/A	Block Size: N/

Showing Page: 6

Cursor Offset: 10388

Selected Block: N/A

Block Size: N/A

pg_hexedit tool, with pg_index_indexrelid_index catalog relation (index)

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

File Edit View Tools Devices Options Help

🕻 🎽 🖾 🖾 🔕 🔍 📿 🔪 🧉		
DataInterpreter 🛛 🖼	16513	
	+0ffse00 01 02 03 04 05 06 07 08 09 04 08 0C 0D 0F 0F 10 11 12 13 14 15 16 17 18 19 14 18 1C 1D 1F 1F 01234567894BCDFF01234567894BCDFF	
Binary 0000000 DEdit		
8 bit 0		
16 bit 512		
32 bit -754974208		
64 bit 2202563248640		
Float E40780268220		
FIOAL -549789508520		
Double 1.0882108339455		
InfoPanel B		
Name: 16513		
Path: /code/		
postgresql/public/./		
data/base/13148		
Size: 40.0 KB		
Access: Read-Write	01680080 97 40 00 60 97 40 00 40 97 40 00 20 97 40 00 00 97 40 00 E0 96 40 00 C0 96 40 00@@@@@@@@	
	016864 C0 95 40 00 A0 95 40 00 80 95 40 00 60 95 40 00 40 95 40 00 20 95 40 00 00 80 41 00 95 40 00	
	016896 E0 94 40 00 00 80 01 00 C0 94 40 00 A0 94 40 00 80 94 40 00 60 94 40 00 40 94 40 00 10 0	
	016928 20 94 40 00 00 80 01 00 00 80 01 00 00 94 40 00 E0 93 40 00 00 80 01 00 C0 93 40 00 A0 93 40 00	
	016960 00 80 01 00 80 93 40 00 60 93 40 00 00 80 01 00 40 93 40 00 20 93 40 00 00 93 40 00	
	01/02460 92 40 00 40 92 40 00 20 92 40 00 00 80 01 00 00 80 01 00 00 80 01 00 00 92 40 00 00 80 01 00 .@.@.@.@.	
	017056 00 80 01 00 E0 91 40 00 C0 91 40 00 A0 91 40 00 80 91 40 00 00 80 01 00 60 91 40 00 40 91 40 00	
	017088 00 80 01 00 20 91 40 00 00 91 40 00 E0 90 40 00 C0 90 40 00 00 80 01 00 A0 90 40 00 80 90 40 00	
	01/120 00 80 01 00 00 80 01 00 60 90 40 00 40 90 40 00 20 90 40 00 00 90 40 00 E0 8F 40 00 00 80 01 00	
	01715200 80 01 00 C0 8F 40 00 A0 8F 40 00 80 8F 40 00 60 8F 40 00 40 8F 40 00 20 8F 40 00 00 8F 40 00	
	017184 00 80 01 00 80 01 00 E0 8E 40 00 C0 8E 40 00 A0 8E 40 00 80 8E 40 00 00 80 01 00 60 8E 40 00	
	017216 40 8E 40 00 20 8E 40 00 00 8E 40 00 E0 8D 40 00 C0 8D 40 00 A0 8D 40 00 00 80 01 00 00 80 01 00 0.eeeeeee.	
	017248 80 8D 40 00 60 8D 40 00 40 8D 40 00 00 80 01 00 00 80 01 00 20 8D 12 224) Ip lep: 0 Ip eff: 0 Ip flags: IP DEAD	
	017280 00 80 01 00 C0 8C 40 00 A0 8C 40 00 80 8C 40 00 60 8C 40 00 00 80 05 EEO PECH 0 PECH 0 PECH 0 PECH 0 PECH 0	
	017344 4C 02 00 00 00 00 00 00 00 00 00 00 00 00	
	017376 4C 02 00 00 4D 02 00 00 00 00 00 00 00 00 00 02 00 E1 00 02 20 00 05 18 00 A3 02 00 00 42 00 00 00 LM	
	017408 4C 02 00 00 4D 02 00 00 00 00 00 00 00 00 00 02 00 E0 00 02 20 00 05 18 00 A2 02 00 00 42 00 00 00 LM	
	017440 4C 02 00 00 00 00 00 00 00 00 00 00 00 00	
	017472 4C 02 00 00 00 00 00 00 00 00 00 00 00 00	
	017504 4C 02 00 00 00 00 00 00 00 00 00 00 00 00	
	017536 4C 02 00 00 00 00 00 00 00 00 00 00 00 00	
	017568 4C 02 00 00 4D 02 00 00 00 00 00 00 00 00 00 02 00 DB 00 02 20 00 05 18 00 9D 02 00 00 42 00 00 00 LM	
	017600 4C 02 00 00 00 00 00 00 00 00 00 00 00 00	
	017632 4C 02 00 00 00 00 00 00 00 00 00 00 00 00	
	017664 4C 02 00 00 00 00 00 00 00 00 00 00 00 00	
	017696 4C 02 00 00 4D 02 00 00 00 00 00 00 00 00 00 02 00 D7 00 02 20 00 05 18 00 99 02 00 00 42 00 00 00 LM	
	017728 4C 02 00 00 4D 02 00 00 00 00 00 00 00 00 00 02 00 D6 00 02 20 00 05 18 00 98 02 00 00 42 00 00 00 LM	
	017760 4C 02 00 00 00 00 00 00 00 00 00 00 00 00	
	017792 4C 02 00 00 00 00 00 00 00 00 00 00 00 00	
	017824 4C 02 00 00 00 00 00 00 00 00 00 00 00 00	
Chaudan Dana 11		Dis she Cit and the
Showing logo 11	Lurson Ottool: LUXZU Curson Voluo: 0 Solostod Plock: N/A	FIOCK SIZOL N/A

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

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

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

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

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; INF0: vacuuming "postgres.public.deltable" INF0: 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

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

HOT UPDATEs

<u>Heap-only tuple optimization</u> 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

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**.

Opportunistic index deletion

- B-Tree also independently cleans up bloat opportunistically, at the level of individual pages
- Postgres 14 greatly improved this mechanism, by <u>making it</u> <u>specifically target non-HOT update bloat</u> 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 <u>perfectly preserve the size of indexes</u> affected by many non-HOT updates
 - Index deduplication (added in Postgres 13) also helps by "soaking up" bursts of duplicates needed for versioning purposes

"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

"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



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.

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+)

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)

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)

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

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

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