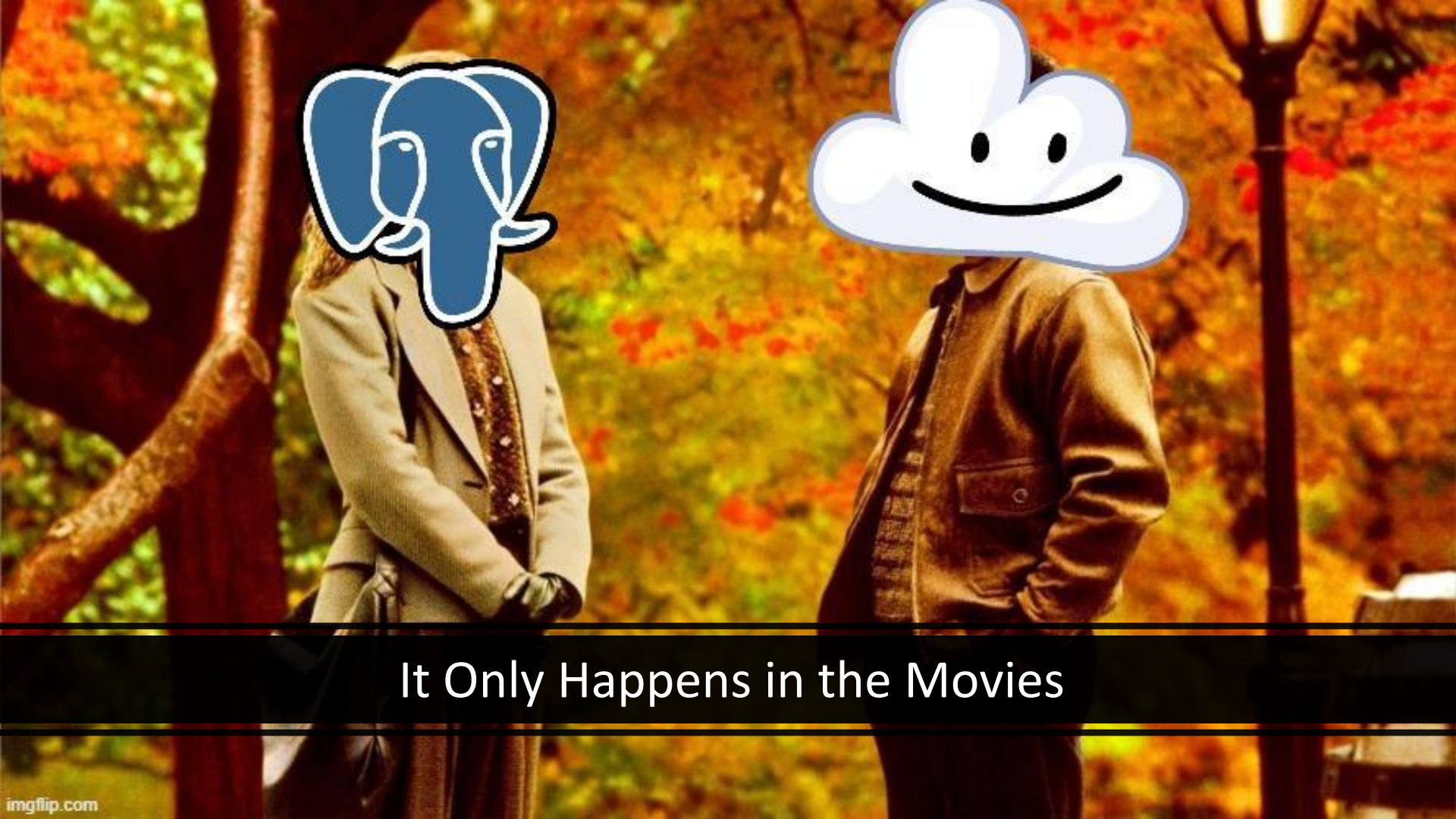
Mayur – DB Specialist@Veeam

When Autovacuum Met FinOps: A Cloud Romance


It Only Happens in the Movies
imgflip.com

Typical DBA/DEV Complaints

- "Autovacuum used to fly on bare-metal, but in the cloud, it feels like it's dragging forever!“
- "I've doubled the max_autovacuum_workers, yet dead tuples just keep stacking up“
- "Every time Autovacuum kicks in on that big table, our application queries start timing out!"

Limits

- Autovacuum throughput is constrained by the autovacuum_vacuum_cost_limit, autovacuum_vacuum_cost_delay and further limited by host restrictions on Cloud.

- Before we delve into tuning Autovacuum, we need some metrics at hand. Autovacuum is known for being I/O-intensive, but just how much strain does it place on your system? Let's explore.

Calculations

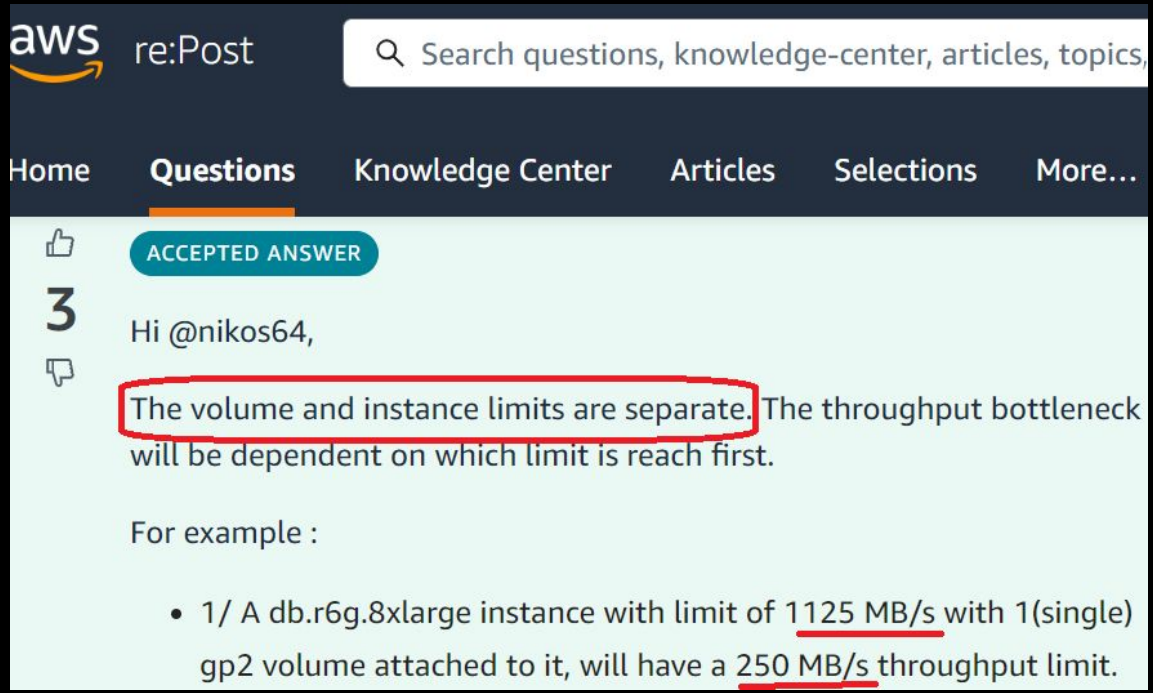
#At most, an autovacuum can do IO as shown below.

Max Autovacuum throughput = (1000/autovacuum_vacuum_cost_delay) * (autovacuum_vacuum_cost_limit/vacuum_cost_page_hit) * 8 KB (default block_size)

#For PG17 default settings:

Max Autovacuum throughput = (1000/2) * (200/1) * 8 KB = 800MB per second

Instance and Storage both limit throughput



Instance throughput limit

Standard instance class : General purpose standard instances of latest graviton series offer better throughput upto 2.5GBps.
Gbps/Mbps here is in Bits hence division by 8 to get GB/s or MB/s.

Model	Core Count	vCPU*	Memory (GiB)	Storage	Dedicated EBS Bandwidth (Gbps)
db.m7g.large	-	2	8	EBS-Only	Up to 10
db.m7g.xlarge	-	4	16	EBS-Only	Up to 10
db.m7g.2xlarge	-	8	32	EBS-Only	Up to 10
db.m7g.4xlarge	-	16	64	EBS-Only	Up to 10
db.m7g.8xlarge	-	32	128	EBS-Only	10
db.m7g.12xlarge	-	48	192	EBS-Only	15
db.m7g.16xlarge	-	64	256	EBS-Only	20

Instance throughput limit

"Up to" is a very vague term. I found another piece of the puzzle in AWS document.

Instance type	Baseline / Maximum bandwidth (Mbps)	Baseline / Maximum throughput (MB/s, 128 KiB I/O)	Baseline / Maximum IOPS (16 KiB I/O)	NVMe	EBS optimization [2]
m7g.medium [1]	315.00 / 10000.00	39.38 / 1250.00	2500.00 / 40000.00	✓	default
m7g.large [1]	630.00 / 10000.00	78.75 / 1250.00	3600.00 / 40000.00	✓	default
m7g.xlarge [1]	1250.00 / 10000.00	156.25 / 1250.00	6000.00 / 40000.00	✓	default
m7g.2xlarge [1]	2500.00 / 10000.00	312.50 / 1250.00	12000.00 / 40000.00	✓	default
m7g.4xlarge [1]	5000.00 /	625.00 / 1250.00	20000.00 /	✓	default

Note

[1] These instances can support maximum performance for 30 minutes at least once every 24 hours, after which they revert to their baseline performance. Other instances can sustain the maximum performance indefinitely. If your workload requires sustained maximum performance for longer than 30 minutes, use one of these instances.

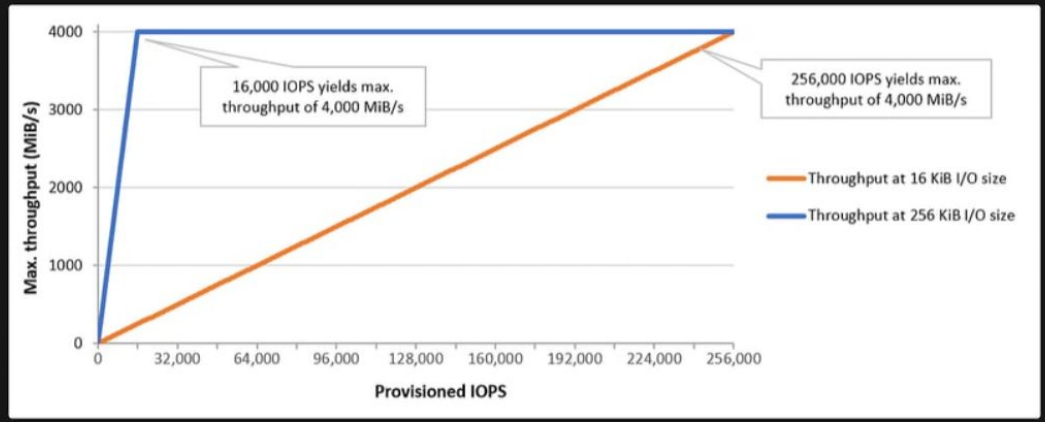
Since maximum performance is guaranteed for only 30 minutes per day, we will focus only on baseline values.

Storage throughput limit

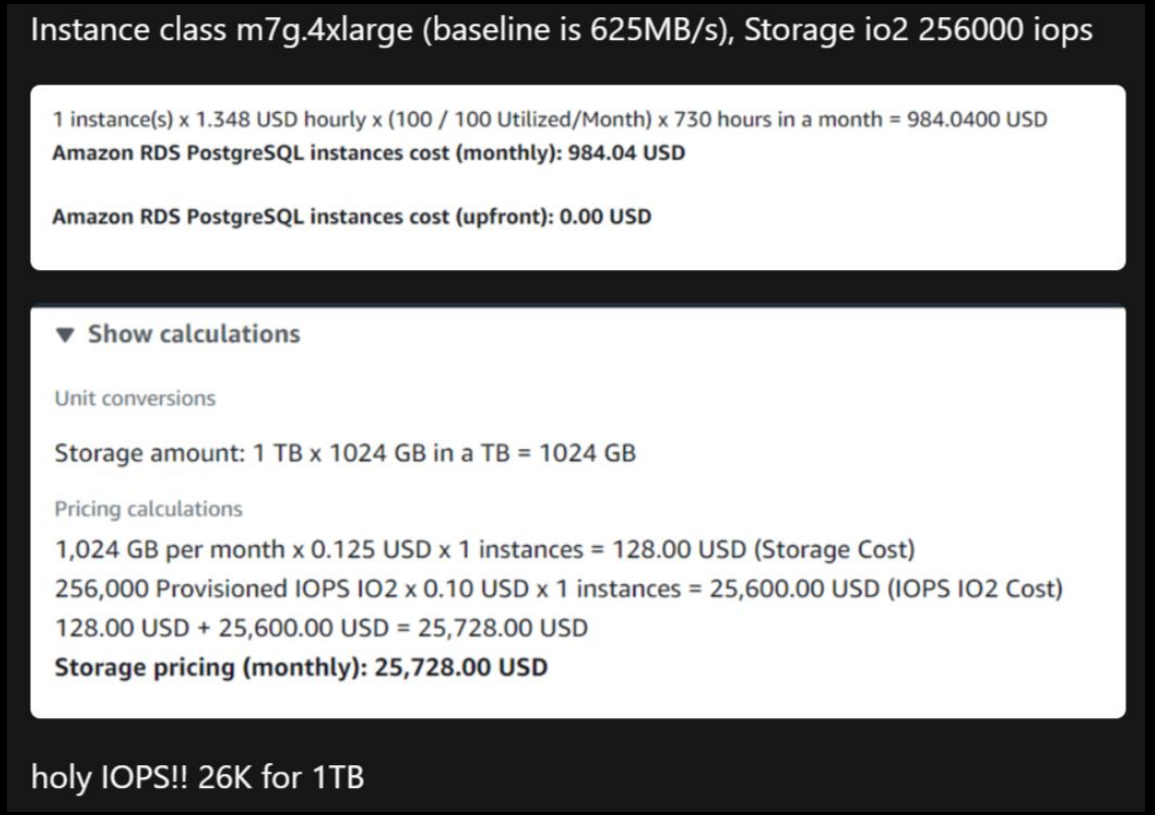
io2 :

- Throughput scales proportionally up to 0.256 MiB/s per provisioned IOPS. Maximum throughput of 4,000 MiB/s can be achieved at 256,000 IOPS with a 16-KiB I/O size and 16,000 IOPS or higher with a 256-KiB I/O size. For DB instances not based on the AWS Nitro System, maximum throughput of 2,000 MiB/s can be achieved at 128,000 IOPS with a 16-KiB I/O size.

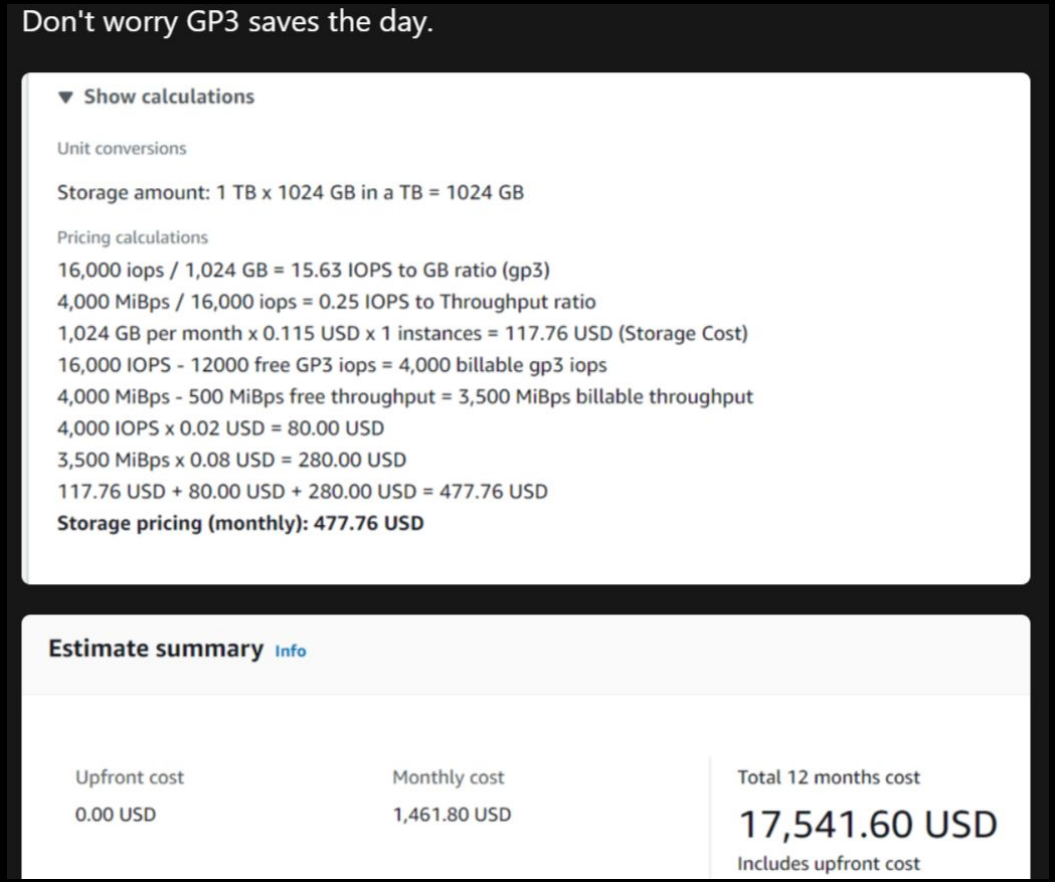
For io2 we can reach up to 500MB/s as we crank up iops knob.



Storage throughput limit



GP3 saves the day



Min. Config for 500MB/s throughput, 1TB db

Cloud	Storage (all SSDs but taking only cost-efficient type)	Compute (Instance class)	Monthly Cost (in US-East)
AZURE (Azure Database for PostgreSQL — Flexible Server)	Premium SSD (5K iops for 500MB/s)	D16ds_v5	$1407
AWS (Amazon RDS – Postgres)	GP3 (16K iops minimum for 500MB/s)	m7g.4xlarge	$1462
GCP (Cloud SQL – Postgres)	Zonal extreme-pd	N2 VM with 64 vCPUs	$2383

Why should you understand costs?

Juggling Cost, Autovacuum Efficiency, and Application Performance

The Low Hanging Fruit -- memory

$$\text{autovacuum\_work\_mem} = \text{LEAST}\left(\frac{\text{Total Memory} - (\text{shared\_buffers} + \text{session\_memory} + \text{reserve\_memory})}{\text{autovacuum\_workers}}, 1\ \text{GB}\right)$$

Example:
Total Memory = 64 GB
shared_buffers = 16 GB (25% of Total Memory)
work_mem = 32 MB
active_sessions = 100
idle_sessions = 500 (assume poorly configured connection pool for more realistic calculations)
idle_session_memory = 10 MB per idle connection
autovacuum_workers = 6

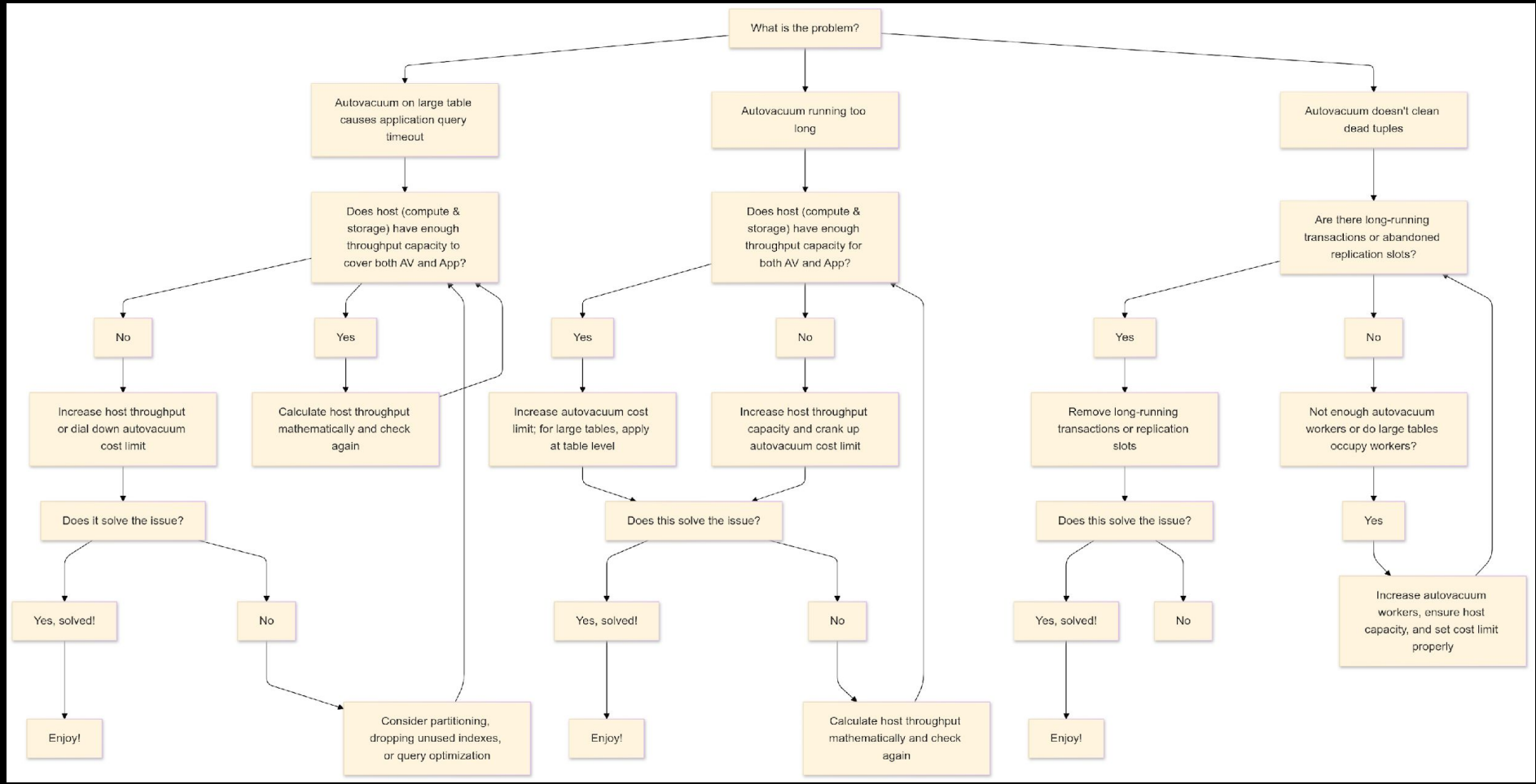
session_memory =(active_sessions*work_mem*hash_mem_multiplier)+ (idle_sessions*idle_session_memory)
Reserve OS Memory = Reserve 20% of Total Memory
autovacuum_work_mem = LEAST(3.9667GB,1GB) =1GB

Reason for autovacuum_work_mem 1GB restriction

postgres / src / include / utils / **memutils.h**

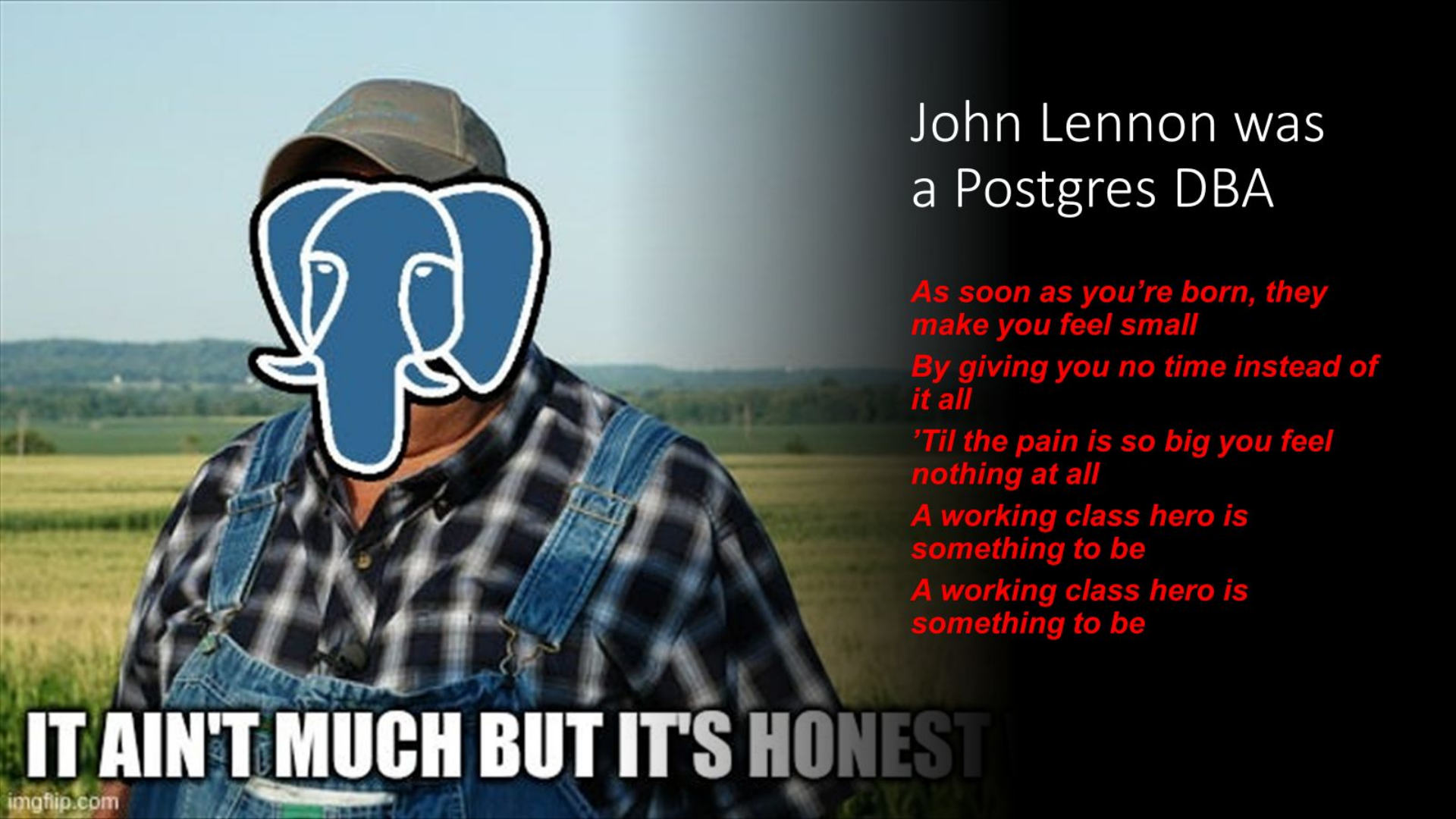
Code | Blame | 209 lines (179 loc) · 6.99 KB

```
38      * compute twice an allocation's size without overflow.
39      */
40     #define MaxAllocSize    ((Size) 0x3fffffff) /* 1 gigabyte - 1 */
41
```


What is the problem?
Autovacuum on large table causes application query timeout
Autovacuum running too long
Autovacuum doesn't clean dead tuples
Does host (compute & storage) have enough throughput capacity to cover both AV and App?
Does host (compute & storage) have enough throughput capacity for both AV and App?
Are there long-running transactions or abandoned replication slots?
No
Yes
Yes
No
Yes
No
Increase host throughput or dial down autovacuum cost limit
Calculate host throughput mathematically and check again
Increase autovacuum cost limit; for large tables, apply at table level
Increase host throughput capacity and crank up autovacuum cost limit
Remove long-running transactions or replication slots
Not enough autovacuum workers or do large tables occupy workers?
Does it solve the issue?
Does this solve the issue?
Does this solve the issue?
Yes
Yes, solved!
No
Yes, solved!
No
Yes, solved!
No
Increase autovacuum workers, ensure host capacity, and set cost limit properly
Enjoy!
Consider partitioning, dropping unused indexes, or query optimization
Enjoy!
Calculate host throughput mathematically and check again
Enjoy!

Budget constraints = Think creatively

- Consider partitioning problematic tables, Size of data and indexes reduces.
- Detect and drop unused indexes (since PG vacuums all indexes).
- Minimize long-running transactions (lower wasteful vacuum runs).
- If it still impacts application performance, you may need to dial back cost limit and delay.
- Always implement an Early Warning System for TXID wraparound (AWS offers a detailed guide on this).
- Last but not the least, upgrade to PG17 for improved vacuuming.



John Lennon was a Postgres DBA

As soon as you're born, they make you feel small

By giving you no time instead of it all

'Til the pain is so big you feel nothing at all

A working class hero is something to be

A working class hero is something to be

Thank you

References:

https://www.percona.com/blog/tuning-autovacuum-in-postgresql-and-autovacuum-internals/

https://calculator.aws/#/

https://azure.microsoft.com/en-us/pricing/details/postgresql/flexible-server

https://cloud.google.com/sql/docs/postgres/pricing

Mayuresh B.

Database Specialist @ Veeam Software | Databases, RDBMS, ...

Database Comedy Blog