Surviving pgvector in Production: A reality check

Miguel Toscano
June 2026

Who am I ?

Data Management Specialist @Google

10+ years breaking databases (Ex-Oracle)

Specializing in Agentic AI & Vector Search

Based in sunny Barcelona

=

Miguel Toscano

The prototype

The prototype

The reality

Our reality

- VM: n2-standard-16 (16 vCPUs, 64GB RAM)
- Storage: 1TB SSD Persistent
- PG 18
- vector: 0.8.2
- Dataset: Synthetic Data
- Embedding Model: nomic-embed-text (768 dimensions)

Prototype	Growth	Production
1K	1M	10M

pgvector: an storage view

- Vector columns write a massive C-array of floats into standard PG Heap
- Uses standard pages, **8kb**
- Relies on **TOAST** for out-of-line storage
- With HNSW, relies on standard **shared_buffers** to cache its graph

```
(768) * 4 bytes (float32) = ~3000 bytes per row
```

The prototype - 1K

shared_buffers = 4GB
maintenance_work_mem = 8GB

table_name	main_heap_size	toast_size	total_size
s1_prototype	192 kB	4000 kB	**4360 kB**

- Reality: Developer is prototyping
- Main heap is tiny
- Vectors are in the TOAST
- 4MB Cache in RAM

The prototype - 1K

shared_buffers = 4GB
maintenance_work_mem = 8GB

table_name	main_heap_size	toast_size	total_size
s1_prototype	192 kB	4000 kB	**4360 kB**

- Reality: Developer is prototyping
- Main heap is tiny
- Vectors are in the TOAST
- 4MB Cache in RAM
- Life is good !

shared_buffers = 4GB
maintenance_work_mem = 8GB

The growth - 1M

table_name	main_heap_size	toast_size	total_size
s2_growth	185 MB	**3906 MB**	4157 MB

- Reality: The app went viral
- Main heap is healthy
- TOAST for a single table is consuming almost the whole cache
- Cache eviction risk

The growth - 1M

shared_buffers = 4GB
maintenance_work_mem = 8GB

table_name	main_heap_size	toast_size	total_size
s2_growth	185 MB	**3906 MB**	4157 MB

- Reality: The app went viral
- Main heap is healthy
- TOAST for a single table is consuming almost the whole cache
- Cache eviction risk
- Warning !

The Production Reality

The Production - 10M

```
shared_buffers = 4GB
maintenance_work_mem = 8GB
```

```
 table_name    | main_heap_size | toast_size | total_size
---------------+----------------+------------+-----------
 s3_production | 1849 MB        | 38 GB      | 41 GB
```

- Reality: Production scale
- Massive consumption of TOAST
- Disk I/O consumption
- Query latency

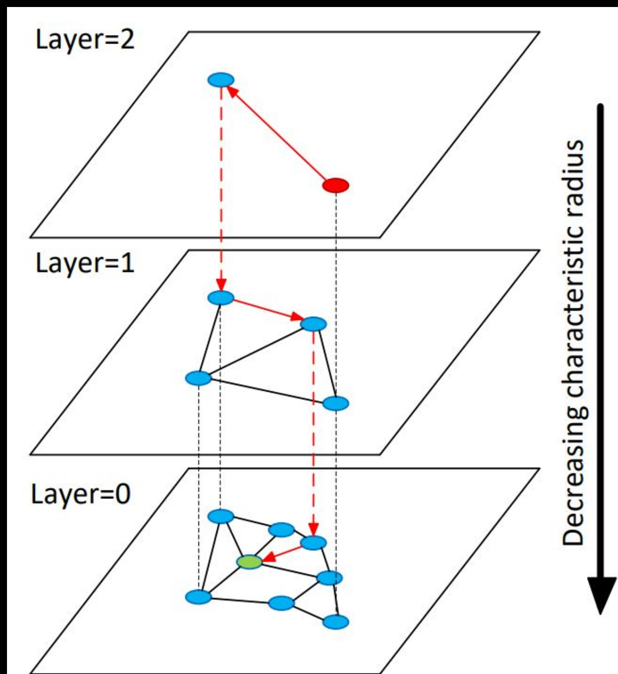
The Production - 10M

shared_buffers = 4GB
maintenance_work_mem = 8GB

```
 table_name   | main_heap_size | toast_size | total_size
--------------+----------------+------------+-----------
s3_production | 1849 MB        | 38 GB      | 41 GB
```

- Reality: Production scale
- Massive consumption of TOAST
- Disk I/O consumption
- Query latency
- Action required !

What is an HNSW index (physically)?



- No predictable left-to-right scan
- Every vector points to its neighbors
- Traversal = Random I/O
- Size really matters

The cost of indexed search?

Scenario	Rows	HNSW Size	Buffer Reads	Latency
S1: Prototype	1K	2568 kB	~253	76 ms
S2: Growth	1M	2427 MB	~158	96 ms
S3: Production	10M	22 GB	207+	186 ms

```sql
SELECT id FROM table_name
ORDER BY embedding <=> '[... 768 dim ...]'
LIMIT 10;
```

The index build tax

rows = 10M
maintenance_work_mem = 8GB

```
-- Scenario 3: The Production
CREATE INDEX idx_s3_cosine ON s3_production USING hnsw (embedding
vector_cosine_ops);

NOTICE:  hnsw graph no longer fits into maintenance_work_mem after 2287073
tuples
DETAIL:  Building will take significantly more time.
HINT:    Increase maintenance_work_mem to speed up builds.

CREATE INDEX
Time: 683046.403 ms (11:23.046)
```

Day 100: Your data changes

```
EXPLAIN (ANALYZE, WAL, BUFFERS) UPDATE
s3_production ... LIMIT 50000;

Update on s3_production
  Buffers: shared hit=64154646
read=257315 dirtied=187391
written=48869
  WAL: records=598790 fpi=167400
bytes=1402014889
...
Execution Time: 98697.330 ms
```

- Updates 0.5% vectors (50K rows)
- MVCC
- HNSW rewire
- WAL

Cache Eviction

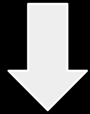
```
SELECT count(*) AS blocks_in_ram
FROM pg_buffercache ... WHERE relname =
'critical_users';

 blocks_in_ram
---------------
          5440
```

Cache Eviction

```
SELECT count(*) AS blocks_in_ram
FROM pg_buffercache ... WHERE relname =
'critical_users';

 blocks_in_ram
---------------
          5440
```



Update 50K vectors

Cache Eviction

```
SELECT count(*) AS blocks_in_ram
FROM pg_buffercache ... WHERE relname =
'critical_users';

 blocks_in_ram
---------------
          5440


SELECT count(*) AS blocks_in_ram_after
FROM pg_buffercache ... WHERE relname =
'critical_users';

 blocks_in_ram_after
---------------------
                   0
```

Cache Eviction

```
SELECT count(*) AS blocks_in_ram
FROM pg_buffercache ... WHERE relname =
'critical_users';

 blocks_in_ram
---------------
          5440


SELECT count(*) AS blocks_in_ram_after
FROM pg_buffercache ... WHERE relname =
'critical_users';

 blocks_in_ram_after
---------------------
                   0
```

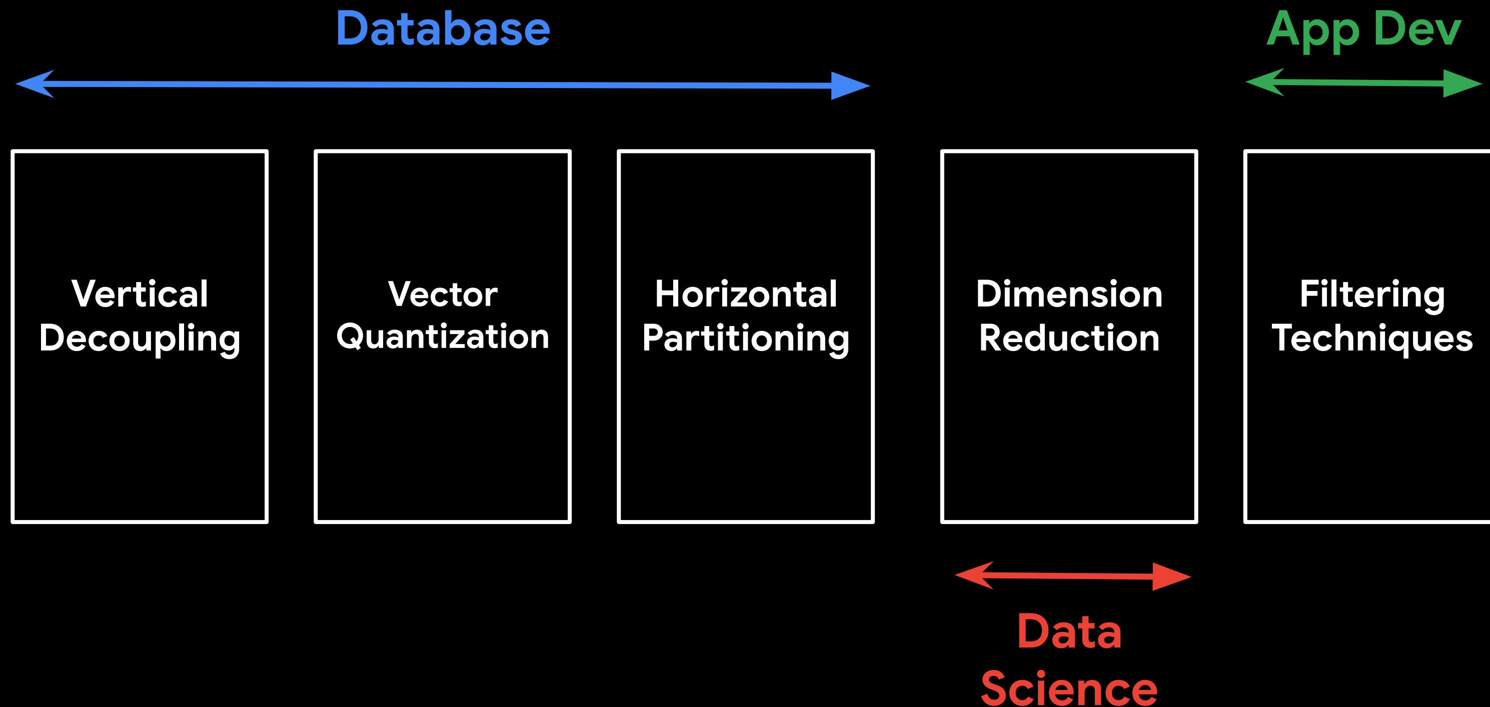
- critical_users was 100% evicted from RAM

The Damage Report

- TOAST Bloat
- Index Size & Query Latency
- Index Building Tax
- The Noisy Neighbor

The Survival Kit

The Survival Kit



Vertical Decoupling

PRODUCTS
id stock_quantity price embedding vector (768)

Vertical Decoupling

PRODUCTS
id stock_quantity price embedding vector (768)

PRODUCTS_META
id stock_quantity price

PRODUCTS_VEC
id embedding vector (768)

Vector Quantization: Shrink the footprint

(32 bits per dimension)

vector(768)
[~3 KB]

[0.12345678] [0.98765432] [-0.45678901] …

Vector Quantization: Shrink the footprint

(32 bits per dimension)

vector(768)
[~3 KB]

[0.12345678] [0.98765432] [-0.45678901] ...

(16 bits per dimension)

halfvec (768)
[~1.5 KB]

[0.1234] [0.9876] [-0.4567] ...

Vector Quantization: Shrink the footprint

(32 bits per dimension)

vector(768)
[~3 KB]

[0.12345678] [0.98765432] [-0.45678901] ...

(16 bits per dimension)

halfvec (768)
[~1.5 KB]

[0.1234] [0.9876] [-0.4567] ...

(1 bit per dimension)

binary
[~96 B]

[1] [1] [0] [1] [0] ...

Vector Quantization: Storage impact at 10M rows

Vector Quantization: The Trade-Off

INFRASTRUCTURE

Table Storage: 41GB -> 19GB -> 2.7GB

Index Storage: 22GB -> 11GB -> 0.3GB

Index Build Time: ~11m -> ~8m -> ~3m

RECALL

- float32: ~98%-99%
- halfvec: ~94%-96%
- Binary: ~70%-80%

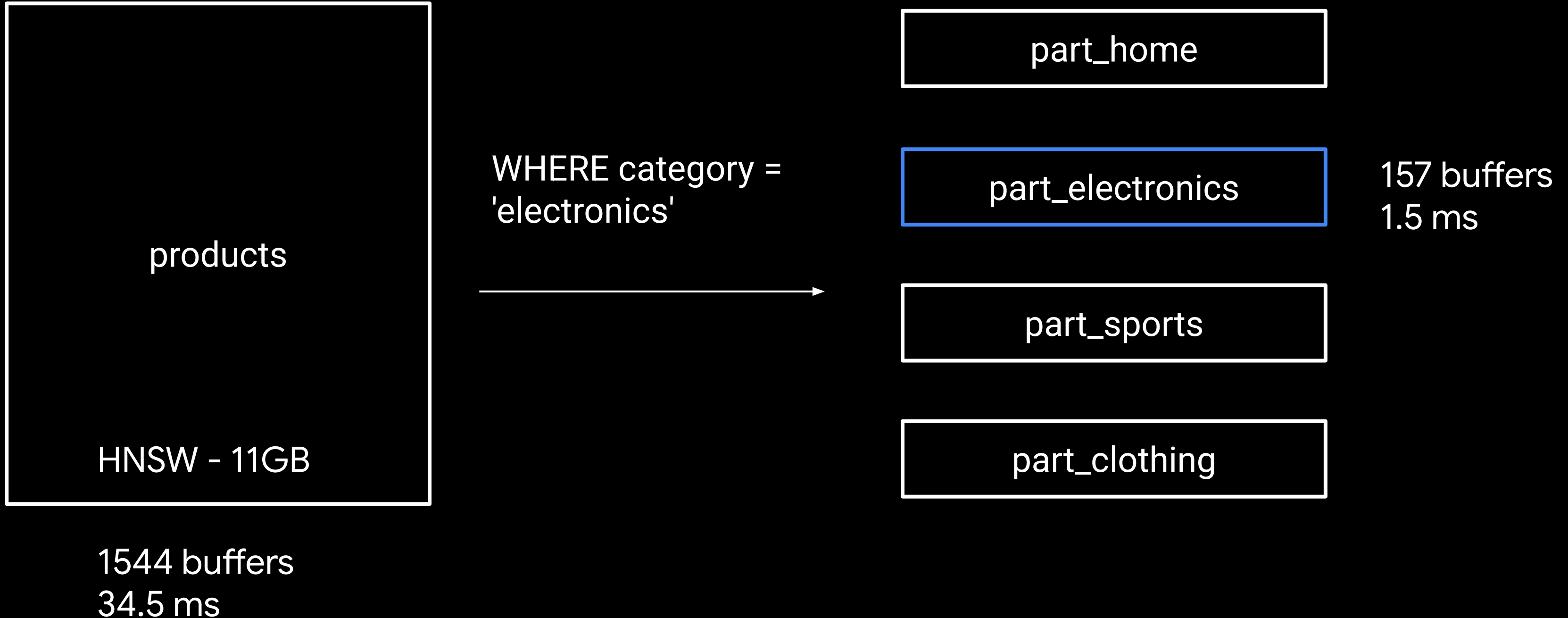
Vector Quantization: The Trade-Off

Benefits	Trade-off
Reduced disk/memory space	Decreased accuracy
Faster queries	Loss of precision
Faster index build	Approximation

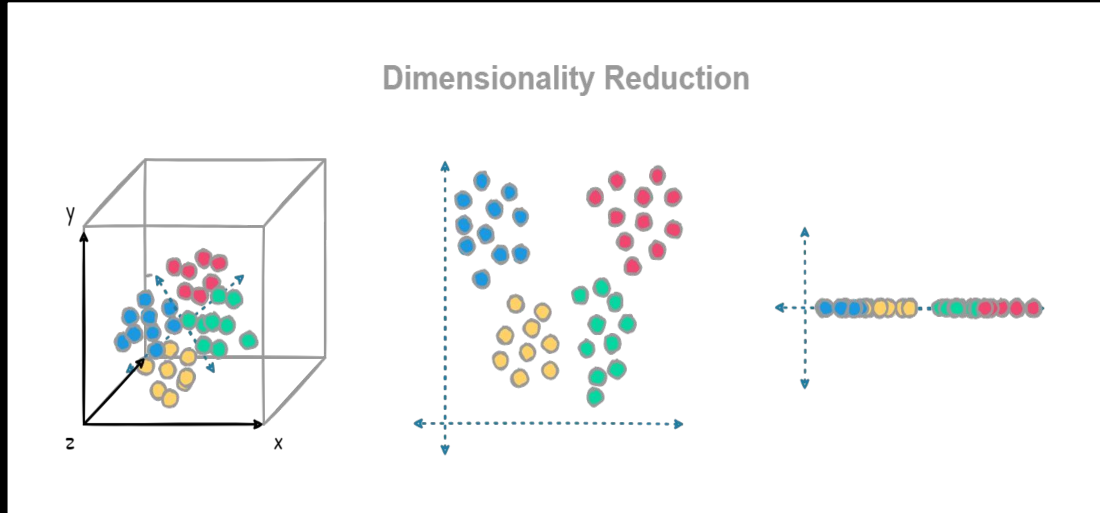
Horizontal Partitioning

1544 buffers
34.5 ms

Horizontal Partitioning



Dimension Reduction

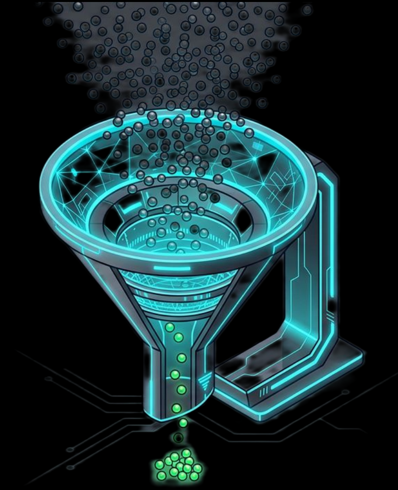


Reduce #dimensions if it doesn't lead to much information loss

There are mathematical techniques that can be employed to reduce the number of dimensions of your vectors while promising that they still retain most of the information content:

- Singular Value Decomposition (SVD)
- Principal Component Analysis (PCA)

Filtering Techniques



Partial ANN Indexes

Create multiple indexes - ANN on the vector column with a index expression on a low-cardinality

Partial ANN indexes can be created for filtering scenarios where only few distinct known values are used for filtering

`Query`

```
SELECT * FROM items WHERE in_stock = true ORDER BY embedding <-> '[3,1,2]' LIMIT 5;
```

`Partial ANN Index`

```
CREATE INDEX ON items USING hnsw (embedding vector_l2_ops) WHERE (in_stock = true);
```

Iterative Index Scans

Scan more of the index until enough results are found!

With approximate indexes, queries with filtering can return less results since filtering is applied after the index is scanned

```
Iterative Index Scan

SELECT * FROM products WHERE price < 50 AND
color = 'red' ORDER BY embedding <=> '[...]'
LIMIT 10;
```

Looking ahead

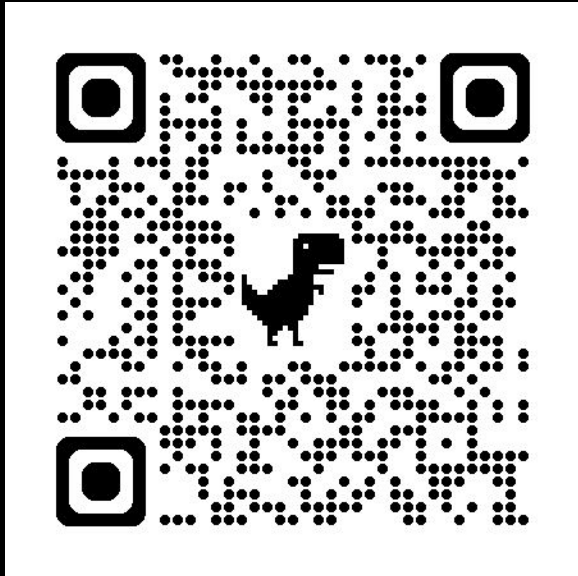
Looking ahead: PG 19

Autoscaling AIO	Concurrently pre-fetches random HNSW index pages to eliminate cold-cache latency
PARALLEL AUTOVACUUM & REPACK CONCURRENTLY	Cleans up vector Write Amplification using parallel workers, or rebuilds bloated tables entirely online
PARTITIONING MANAGEMENT	Native DDL to split or merge vector partitions as your catalog grows

Key Takeaways

Stop the TOAST Bloat		Vector Quantization + dimension reduction
Protect your cache		Vector Quantization + Partial Indexes
Isolate the updates		Vertical decoupling
Partition the graph		Horizontal Partitioning

Thank you.

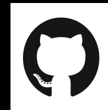


Miguel Toscano

Data Management Specialist
Google Cloud

mtoscano@google.com

www.linkedin.com/in/mtoscanorodero


github.com/mtoscano84