Inside pgvector

How PostgreSQL Stores, Indexes & Manages High-Dimensional Data

Narrated by: Jeevan

April 2026

Why This Talk

Every team is adding vector search. Few plan for what happens next.


```
Month 1:  "Let's add semantic search!"
          → pgvector, 100K vectors, works great
✅

Month 4:  "Scale to all our docs"
          → 10M vectors, still fine ✅

Month 8:  "Enterprise rollout"
          → 100M vectors, RAM bill explodes 🤯

Month 10: "Maybe we need a vector DB?"
          → now syncing two databases forever
🔄
```

This talk: What happens inside Postgres when you store and search vectors — so you decide before month 8.

Chapter 1: What's an Embedding?

Text → Numbers that capture meaning

```
"I love pizza"     → [0.2, 0.8, 0.1, ... 384 numbers]
"Pizza is great"   → [0.3, 0.7, 0.2, ... 384 numbers]
"The sky is blue"  → [0.9, 0.1, 0.8, ... 384 numbers]
```

Key insight: Similar meanings → Similar numbers!

Like GPS coordinates for meaning — similar text ends up nearby in 384-dimensional space.

🟦🟫 Quick Demo: Semantic Search

Pre-seeded: 12 PostgreSQL docs in `docs_demo` (384d embeddings)

```
SELECT id, content FROM docs_demo;
```

Now search by meaning, not keywords:

```
python scripts/embed_query.py "how to find and fix slow queries"
```

```
SELECT content,
       embedding <=> '<paste_vector>'::vector AS distance
FROM docs_demo
ORDER BY distance
LIMIT 3;
```

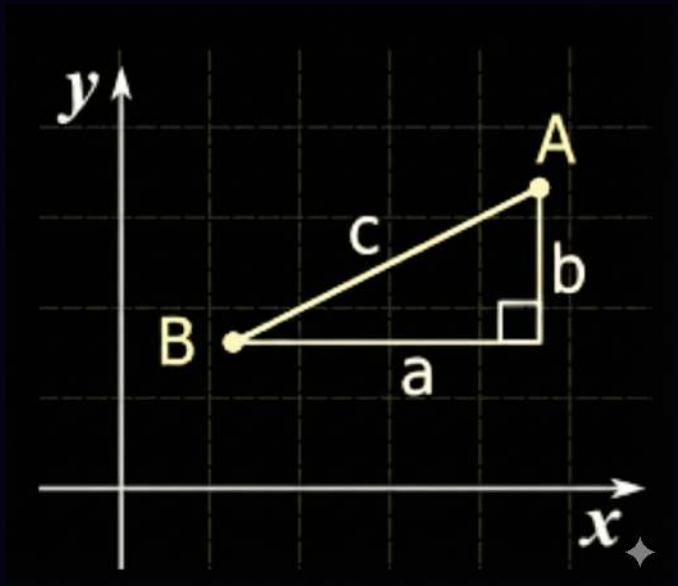
Zero keyword overlap — pure semantic match ✨

Chapter 2: Distance Operators

How do we find similar vectors? Distance functions.

Operator	Name	Use When
`<->`	L2 (Euclidean)	Absolute distance matters (images, spatial)
`<=>`	Cosine	Direction matters (text — most common)
`<#>`	Inner Product	MaxIP retrieval (recommendations)

Most common for text: Cosine `<=>` — normalizes for you, so vector magnitude doesn't matter.



Chapter 3: Where Do Vectors Live in Postgres?

```
CREATE EXTENSION IF NOT EXISTS vector;
CREATE TABLE docs (
    id serial PRIMARY KEY, content text, embedding vector(1024)
);
-- 50,000 docs with 1024d embeddings preloaded
```

```
SELECT pg_size_pretty(pg_relation_size('docs')) AS heap_size;
```

The math: 50,000 × 1024 dims × 4 bytes = ~200 MB of raw embeddings

But heap is only ~27 MB. Where's the other ~173 MB? 🤔

🟦 The Mystery Revealed: TOAST!

```sql
SELECT c.relname,
    pg_size_pretty(pg_relation_size(c.oid)) AS heap_size,
    pg_size_pretty(pg_relation_size(c.reltoastrelid)) AS toast_size,
    pg_size_pretty(pg_total_relation_size(c.oid)) AS total_size
FROM pg_class c WHERE c.relname = 'docs';
```

Aha! 🐘 TOAST = The Oversized-Attribute Storage Technique

Large columns → moved to a separate table automatically.

- Each embedding: ~4,100 bytes
- Heap: ~27 MB (row headers + content text)
- TOAST: ~260 MB (70% of data!)
- Total: ~371 MB

1024d vectors (~4KB) > 2KB tuple threshold → out-of-line Every vector read = extra TOAST I/O

- 384d (~1.5 KB) → stays inline ✅
- 1024d (~4 KB) → TOASTed, slower ❌

Bigger isn't always better 🏋️

```
EXPLAIN ANALYZE
SELECT id, content,
       embedding <=> (SELECT embedding FROM docs WHERE id = 75) AS distance
FROM docs ORDER BY distance LIMIT 5;
```

Observe: Seq Scan — reads all 50,000 rows, ~500-2000ms, ~300,000 buffer reads (TOAST!).

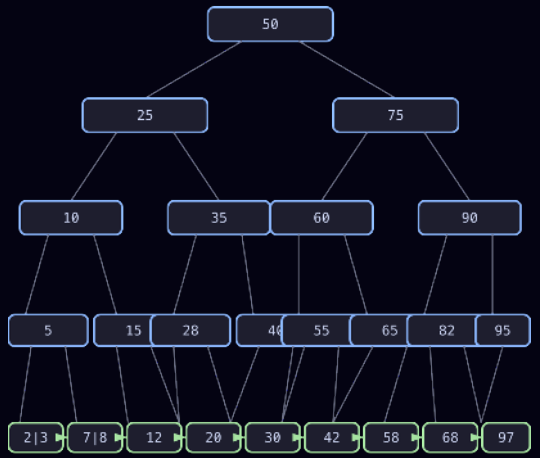
O(N × dimensions) for every single query.

Why B-Trees Can't Help

Regular queries:

```
SELECT * FROM users WHERE id = 42;
```

B-tree → binary search → O(log n).



B+ Tree — sorted keys, O(log n) lookup
Leaf-linked for range scans. Binary search works
because there's a natural order.

Vector queries:

```
SELECT * FROM docs
ORDER BY embedding <=> query LIMIT 10;
```

B-trees need a linear ordering. Vectors don't have one.

```
[0.23, -0.89, 0.45, ...]
[0.91, 0.03, -0.44, ...]
Sort these? By which number? 🤷
```

We need Approximate Nearest Neighbor (ANN) indexes.

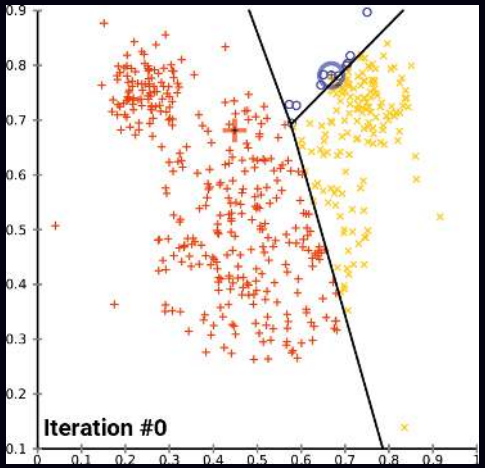
Finding 9 out of 10 true best matches is good enough.

IVFFlat: Cluster & Search

IVFFlat = Inverted File Index with Flat Storage

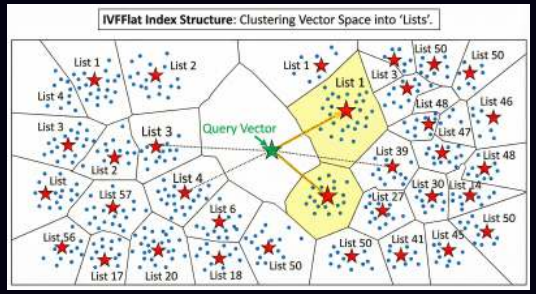
Build: Divide vectors into clusters (k-means)

"Supermarket aisles — dairy, snacks, spices"
🛒



Query: Search only the nearest cluster(s)

"Need butter? Go to dairy aisle, skip the rest" 🏃



IVFFlat in Action

```
SET maintenance_work_mem = '512MB';
CREATE INDEX docs_ivfflat_idx
ON docs USING ivfflat (embedding vector_cosine_ops)
WITH (lists = 200);
ANALYZE docs;
```

- `lists = 200` (≈ √50k) — number of clusters
- `vector_cosine_ops` — matches the `<=>` operator

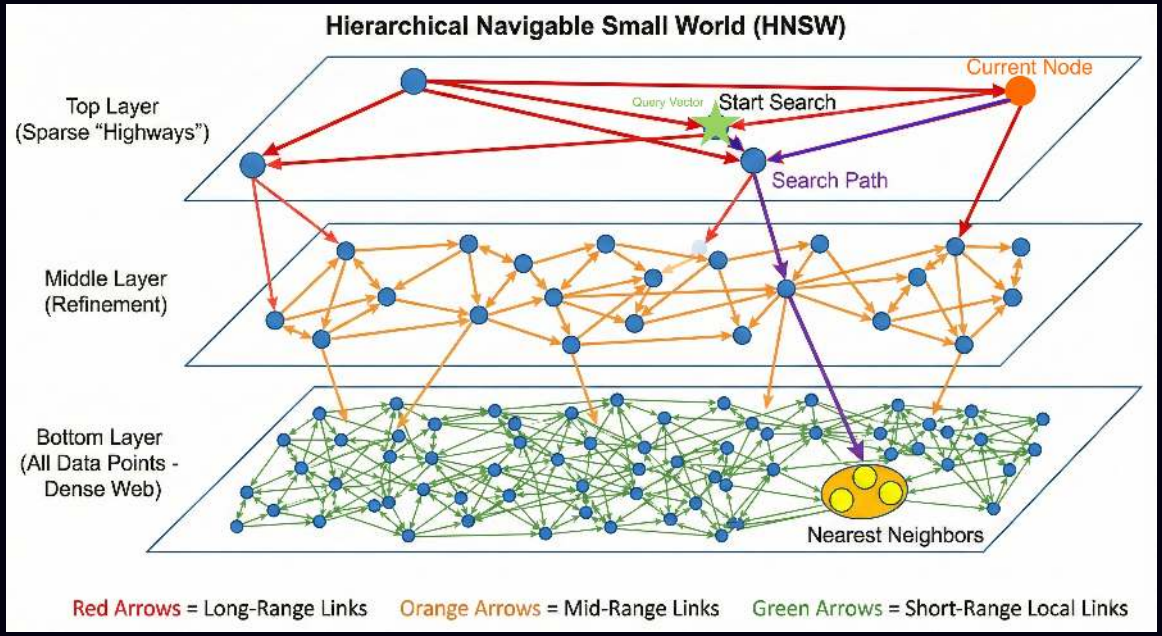
```
-- Forcing index for demo clarity.
-- In production, planner picks this with LIMIT + ANALYZE.
SET enable_seqscan = off;

EXPLAIN (ANALYZE, BUFFERS)
SELECT id, content,
       embedding <=> (SELECT embedding FROM docs WHERE id = 1) AS distance
FROM docs ORDER BY distance LIMIT 5;
```

Result: Index Scan — ~10-20ms, ~1,000 buffers. 50-100x faster.

HNSW: Multi-Layer Graph Search

HNSW = Hierarchical Navigable Small World — higher recall than IVFFlat.



Like GPS — highways first, then local roads to the exact address.

HNSW in Action

```
DROP INDEX docs_ivfflat_idx;
CREATE INDEX docs_hnsw_idx
ON docs USING hnsw (embedding vector_cosine_ops)
WITH (m = 16, ef_construction = 200);
```

- `m = 16`: connections per node — `ef_construction = 200`: build-time search depth

```
SET hnsw.ef_search = 40;
EXPLAIN (ANALYZE, BUFFERS)
SELECT id, content,
    embedding <=> (SELECT embedding FROM docs WHERE id = 1) AS distance
FROM docs ORDER BY distance LIMIT 5;
```

Method	Time	Speedup	Buffers	Recall
Seq Scan	500-2000ms	1x	~300,000	100%
IVFFlat	10-20ms	50-100x	~1,000	~95%
HNSW	5-15ms	70-150x	~500	~99%

Build: IVFFlat ~3s vs HNSW ~77s 🐢 (50k rows, 1024d)

🟦 ⚠️ Always Use LIMIT!

Without LIMIT, Postgres won't use ANN indexes!

```sql
-- BAD: Sequential scan even with indexes!
SELECT id, content FROM docs
ORDER BY embedding <=> (SELECT embedding FROM docs WHERE id = 1);

-- GOOD: ANN index kicks in
SELECT id, content FROM docs
ORDER BY embedding <=> (SELECT embedding FROM docs WHERE id = 1)
LIMIT 10;
```

Why? ANN indexes find top K nearest — no K = no early stopping = full scan.

Forgetting LIMIT is like ordering everything on the menu just to pick one dish 🤦

Chapter 5: Production Reality

MVCC Bloat: UPDATE copies the entire row (including 4KB embedding!). 5000 metadata updates = 20 MB dead TOAST data.

Fix: Separate embedding table

```
CREATE TABLE documents (
  id serial, title text, content text, metadata jsonb
);
CREATE TABLE embeddings (
  doc_id int REFERENCES documents(id),
  embedding vector(1024), content_hash text
);
-- Metadata updates no longer duplicate embeddings
UPDATE documents SET metadata = '{"views": 100}' WHERE id = 1;
```

More tips:

- Use smaller models (384d stays inline, 1024d gets TOASTed)
- Increase `maintenance_work_mem` for index builds
- `ANALYZE` after bulk inserts (IVFFlat needs statistics)
- Create indexes AFTER bulk loading, not during

🟦 Key Takeaways

pgvector handles 1k-10M vectors in production. No specialized DB needed.

Storage: 1024d vectors get TOASTed (~4KB > tuple threshold). Smaller models = less I/O.

Indexes: HNSW for production reads (~99% recall), IVFFlat for batch/writes (~95%). Always use LIMIT.

Production: Separate embedding tables. Watch MVCC bloat. Test filtered search early.

When to look beyond pgvector: Multi-node sharding at 100M+, sub-ms latency requirements, or zero-downtime streaming updates.

Postgres can do semantic search. 🎉

The End

Now go forth and vector! 🚀

Questions?

📚 Advanced topics in the same repo: quantization, DiskANN, hybrid search, architecture trade-offs → `vector_search_fundamentals.md` & `vector_storage_at_scale.md`

📧 jeevan.dc24@alumni.iimb.ac.in

🌐 noobj.me

Slides & Code:

Appendix

Backup slides — if we have time

The Filtered Search Trap

In production, you rarely search the entire table:

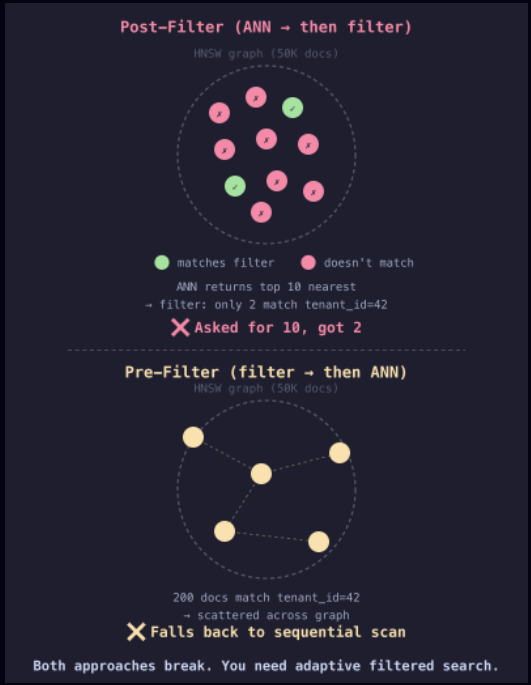
```
SELECT * FROM docs
WHERE tenant_id = 42
  AND category = 'electronics'
ORDER BY embedding <=> query_embedding
LIMIT 10;
```

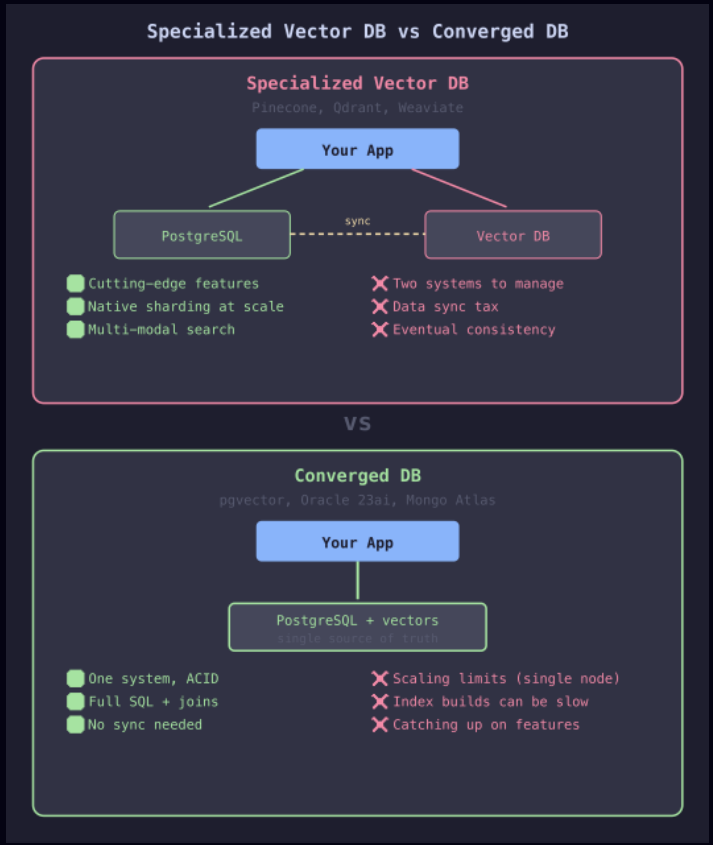
Problem: Vector indexes are blind to metadata filters.

Fixes:

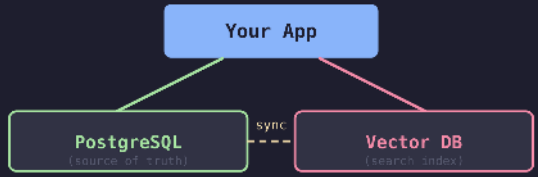
- `iterative_scan` — keep scanning until enough filtered results
- Partial indexes — separate HNSW per category
- Table partitioning — one partition per tenant

#1 gotcha in production vector search.





Start with the existing database. Migrate only when it's outgrown.



What goes wrong:

❌ Vector DB write fails
→ stale search results, users see old data

❌ Doc deleted from Postgres only
→ ghost results, deleted content searchable

❌ Embedding model updated
→ must re-embed everything in both systems

❌ User permissions change
→ vector DB returns unauthorized results

Two databases = eventual consistency.
Converged DB = one source of truth.

It's a genuine trade-off, not a clear winner.

Mitigations exist (CDC, outbox pattern) but add cost.

The RAM Wall

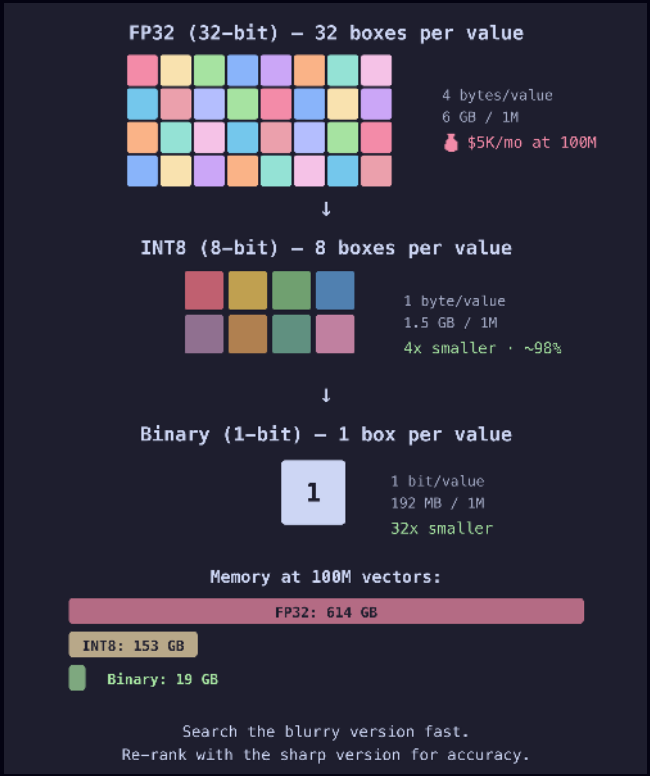
```
Per vector:  1536 dims × 4 bytes = 6 KB
```

Scale	Raw Vectors	+ HNSW (~50%)	Approx. RAM Cost
1M	6 GB	~9 GB	~$50/mo
10M	61 GB	~92 GB	~$500/mo
100M	614 GB	~920 GB	~$5,000+/mo
1B	6.1 TB	~9.2 TB	💀

64 GB → 920 GB isn't 15x cost — it's 30-50x operational complexity.

Quantization: Compress Smartly

Full precision isn't needed for searching. Only for final ranking.



Quantization Trade-offs

Method	Compression	Recall (w/ re-rank)	Speed	Training?
FP32 (baseline)	1x	100%	1x	No
FP16 (half)	2x	~99.9%	~1.5x	No
Scalar INT8	4x	~98-99%	~2-3x	No
Product (PQ)	8-64x	~95-99%	~5-10x	Yes
Binary (BQ)	32x	~92-96%	~15-30x	No

Start with FP16 (free 2x, zero recall loss). Graduate to BQ + re-rank when 32x compression is needed.

FP16 (halfvec) — 2x compression, pgvector native:

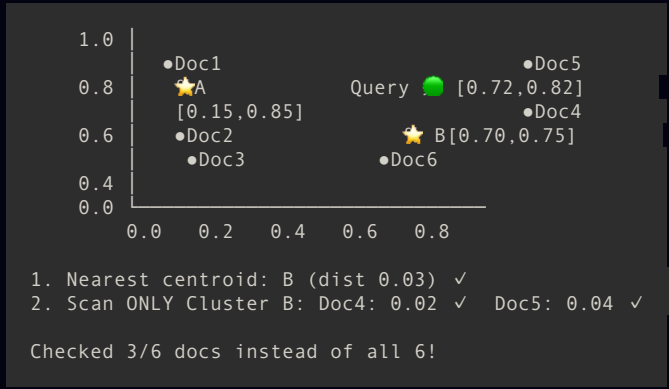
```
ALTER TABLE docs ADD COLUMN embedding_half halfvec(1536);
UPDATE docs SET embedding_half = embedding::halfvec(1536);
CREATE INDEX ON docs USING hnsw (embedding_half halfvec_cosine_ops);
```

Binary Quantization — 32x compression, pgvector native:

```
ALTER TABLE docs ADD COLUMN embedding_bit bit(1536);
UPDATE docs SET embedding_bit = binary_quantize(embedding)::bit(1536);
CREATE INDEX ON docs USING hnsw (embedding_bit bit_hamming_ops);

-- BQ coarse pass + FP32 re-rank
WITH candidates AS (
  SELECT id, embedding FROM docs
  ORDER BY embedding_bit <~> binary_quantize($query)::bit(1536)
  LIMIT 200
)
SELECT id, content FROM candidates
ORDER BY embedding <=> $query LIMIT 10;
```

IVFFlat Search Process



🟦 IVFFlat Build: K-Means Clustering

```
Doc1: [0.1, 0.2, 0.9]   Doc4: [0.7, 0.8, 0.1]
Doc2: [0.2, 0.3, 0.8]   Doc5: [0.75, 0.85, 0.05]
Doc3: [0.15, 0.25, 0.85] Doc6: [0.65, 0.75, 0.15]
```

⚠️ Must load ALL vectors into memory first!

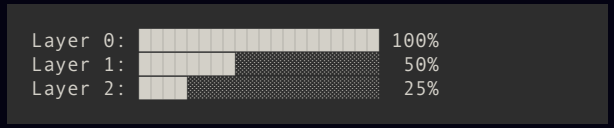
Process: Pick K random centroids → assign vectors to nearest → recalculate centroids → repeat until convergence.

```
Final:
  Centroid 1: [0.15, 0.25, 0.85] → Doc1, Doc2, Doc3
  Centroid 2: [0.70, 0.80, 0.10] → Doc4, Doc5, Doc6
```

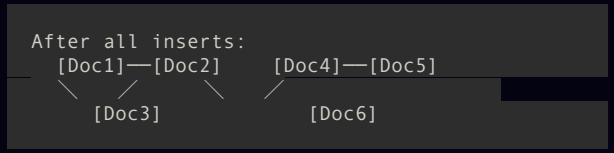
HNSW Build: Incremental Graph

✓ NO need to load all data at once — inserts one vector at a time.

Layer assignment (exponential decay):



Insert = search for nearest neighbors → connect → prune to M max



```sql
-- Are indexes being used?
SELECT indexrelname, idx_scan, idx_tup_read
FROM pg_stat_user_indexes WHERE relname = 'docs';

-- Dead tuple bloat
SELECT relname, n_live_tup, n_dead_tup,
  round(100.0 * n_dead_tup / NULLIF(n_live_tup + n_dead_tup, 0), 1) AS dead_pct
FROM pg_stat_user_tables WHERE relname = 'docs';

-- Index sizes
SELECT indexrelname, pg_size_pretty(pg_relation_size(indexrelid)) AS size
FROM pg_stat_user_indexes WHERE relname = 'docs';
```

References

- pgvector: https://github.com/pgvector/pgvector
- HNSW: https://www.pinecone.io/learn/series/faiss/hnsw/
- IVFFlat: https://milvus.io/docs/ivf-flat.md
- TOAST: https://www.postgresql.org/docs/current/storage-toast.html
- MVCC: https://www.postgresql.org/docs/7.1/mvcc.html