Inside pgvector

How PostgreSQL Stores, Indexes & Manages High-Dimensional Data

December 19, 2025

Narrated by: Jeevan

Our Journey Today

1. Embeddings & Distance

- What are embeddings?
- How to compare vectors?
- Distance operators

2. Storage Deep Dive

- Where vectors live in Postgres
- TOAST behavior (1024d = 4KB!)
- Why size matters

3. Search Without Indexes

- Semantic search basics
- Sequential scan performance
- The O(N) problem

4. ANN Indexes

- IVFFlat: k-means clustering
- HNSW: hierarchical graphs
- 50-150x speedup!

5. Production Reality

- MVCC bloat problem
- Separate embedding tables
- Index strategy & tuning
- Best practices for scale

📘 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!

It's like teaching your database that "pizza" and "delicious" are besties 🍕❤️

In other words: We're teaching computers that "pizza" and "great" go together. 🍕

🟦🎯 Let's See It In Action

First: How similar are two pieces of text?

```
python demo.py compare
```

Now Let's Store These Embeddings

Step 1: Add some texts to Postgres

```
python demo.py seed
```

Search Your Data

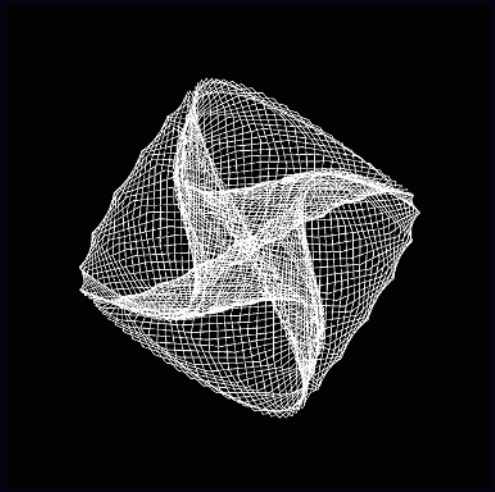
Step 2: Find similar content

```
python demo.py search
```

What Just Happened? Let's Look Inside

Postgres stored our embeddings as vectors

```
-- See all documents
SELECT id, content FROM demo;

-- Check embedding dimensions
SELECT id, content,
array_length(embedding::float4[], 1)
AS dimensions
FROM demo LIMIT 3;

-- Peek at first 5 numbers
SELECT id, content,
       (embedding::float4[])[1:5] AS
first_5_values
FROM demo LIMIT 3;
```



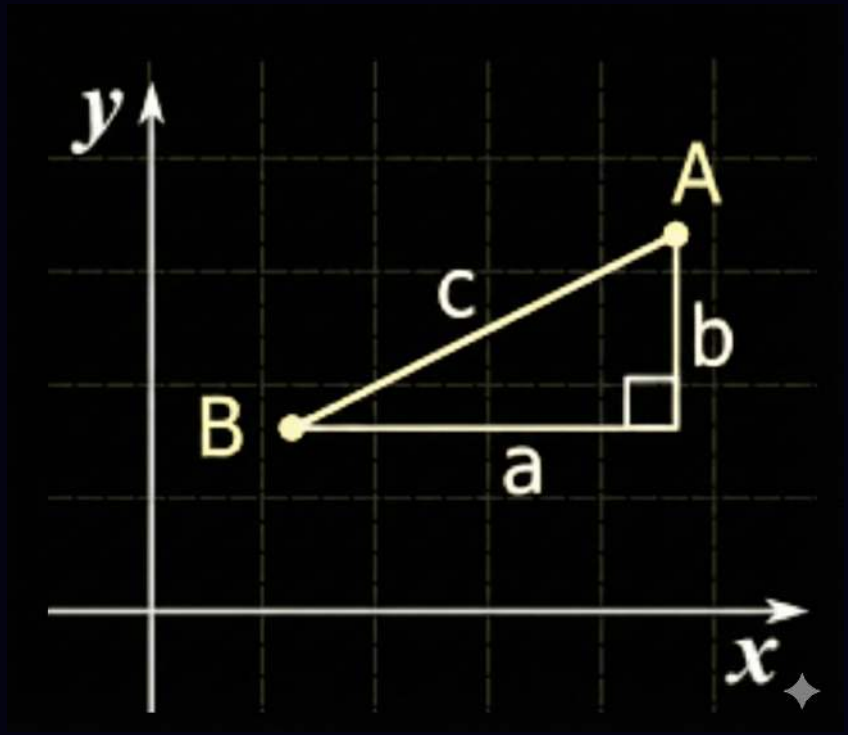
The magic: Text → 384 numbers → Semantic search ✨

Why higher dimenions matter$_2$?

🟦 Chapter 2: How Do We Compare Vectors?

Now that we have embeddings stored, how do we find similar ones?

We need a way to compare and that's distance functions!



Three Distance Operators

Operator	Name	Formula	Use When
`<->`	L2 (Euclidean)	$\sqrt{(\Sigma(a-b)^2)}$	Absolute distance matters (images)
`<=>`	Cosine	$1 - (a \cdot b)/(\|a\|\|b\|)$	Direction matters (text, most common)
`<#>`	Inner Product	$-(a \cdot b)$	Pre-normalized vectors

Simple analogies:

- L2: Two people in a city - how many blocks apart?
- Cosine: Two people facing directions - how similar?
- Inner Product: Two arrows - how aligned?

Most common: Cosine `<=>` (works best for text embeddings)

Think of it as three ways to measure how much your vectors "vibe" together ✨

Example: Finding Similar Documents

```sql
-- Find similar docs using cosine distance
SELECT
  content,
  embedding <=> (
    SELECT embedding
    FROM demo
    WHERE content = 'Inflation is high'
    LIMIT 1
  ) AS distance
FROM demo
ORDER BY distance
LIMIT 5;
```

Key: Lower distance = more similar (always use LIMIT!)

Cosine Distance varies from 0 to 2 and the cosine similarity angle varies from -1 to 1

🟦 Chapter 3: How Are Vectors Stored in Postgres?

We know how to compare vectors, but where does Postgres actually store them?

Let's explore step by step...

Step 1: Create the Table

```
CREATE EXTENSION IF NOT EXISTS vector;

CREATE TABLE IF NOT EXISTS docs (
    id serial PRIMARY KEY,
    content text,
    embedding vector(1024)
);
```

Step 2: Load Demo Data

```
# Generate 50,000 documents with embeddings
python generate_demo_embeddings.py
```

This will:

- Generate realistic random text using Faker
- Create 1024-dimensional embeddings (we will demonstrate TOAST!)
- Insert 50k rows into docs table

I have preloaded this data to save time

Step 3: Inspect the Data

```sql
-- How many documents?
SELECT COUNT(*) FROM docs;
-- Expected: 50000

-- See sample content
SELECT id, LEFT(content, 50) AS content_trunc FROM docs LIMIT 3;

-- Check embedding size
SELECT
    id,
    LEFT(content, 50) AS content_trunc,
    pg_column_size(embedding) AS embedding_bytes,
    array_length(embedding::float4[], 1) AS dimensions
FROM docs LIMIT 3;
```

Expected: ~4 KB per 1024-dim embedding

🟦 Step 4: Where Is the Data Stored?

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

Let's do the math:

- 50,000 docs × 1024 dimensions × 4 bytes = 200 MB

But we only see:

- Heap size: ~11 MB

Wait... where's the other ~189 MB? 🤔

🟦 The Mystery Revealed: TOAST!

```sql
-- Check individual row and embedding size
SELECT
    id,
    pg_column_size(embedding) AS embedding_bytes,
    pg_column_size(docs.*) AS row_bytes
FROM docs
LIMIT 3;

-- Check TOAST table size
SELECT
    c.relname AS table_name,
    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! 🦴

- Each embedding: ~4,100 bytes (> 2KB threshold!)
- Heap size: ~11 MB (just row headers)
- TOAST size: ~260 MB (94% of data is here!)
- Total: ~276 MB

Key insight: 1024d embeddings go to TOAST (> 2KB threshold) ✓

Chapter 4: What About Embedding Sizes?

We saw TOAST in action with 1024d. But what if we used smaller embeddings?

Let's compare different embedding sizes...

Our demo (50k docs, 1024d):

```
Heap:  ████▒▒▒▒▒▒▒▒▒▒▒▒  ~50 MB
TOAST: ████████▒▒▒▒▒▒▒▒  ~200 MB
```

✓ Most embeddings in TOAST (demonstrates the problem!)

If we used 384d instead:

```
Heap:  ████████████▒▒▒▒  ~50 MB
TOAST: ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒   ~0 MB
```

✓ Embeddings stay in heap (no extra I/O!)

Why does this matter?

TOAST = extra I/O

- Every query reads from 2 places
- More disk seeks
- Slower performance

Recommendation: Use smaller models when possible!

- 384d: MiniLM, E5-small (~1.5 KB)
- 512d: E5-base (~2 KB, borderline)
- 1024d: BGE-large (~4 KB, TOASTed)

Bigger isn't always better. Sometimes it's just... slower 🐌

🟦 What is TOAST?

```
TOAST = "The Oversized-Attribute Storage Technique"
```

Not this kind of toast: 🍞

Analogy:

It's like getting up to fetch a file every single time. Annoying, right?

Impact: TOAST = extra I/O on every read!

Chapter 5: The Baseline - Search Without Indexes

Now we understand storage. Let's actually search for similar documents!

First, let's see how it works WITHOUT any indexes...

```
-- Get a random doc and check the content
SELECT content FROM docs WHERE id = 97;

-- Find similar docs
SELECT id, LEFT(content, 150) as contect_trunc,
       embedding <=> (
         SELECT embedding FROM docs WHERE id = 97
       ) AS distance
FROM docs
ORDER BY distance
LIMIT 5;
```

Notice: Results are semantically similar! But how fast is it?

Postgres checking every single row like:

"Is this the one? No. Is THIS the one? No..."

Chapter 6: The Problem - Sequential Scan Performance

The search works, but is it fast? Let's measure...

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

The Problem

Observe:

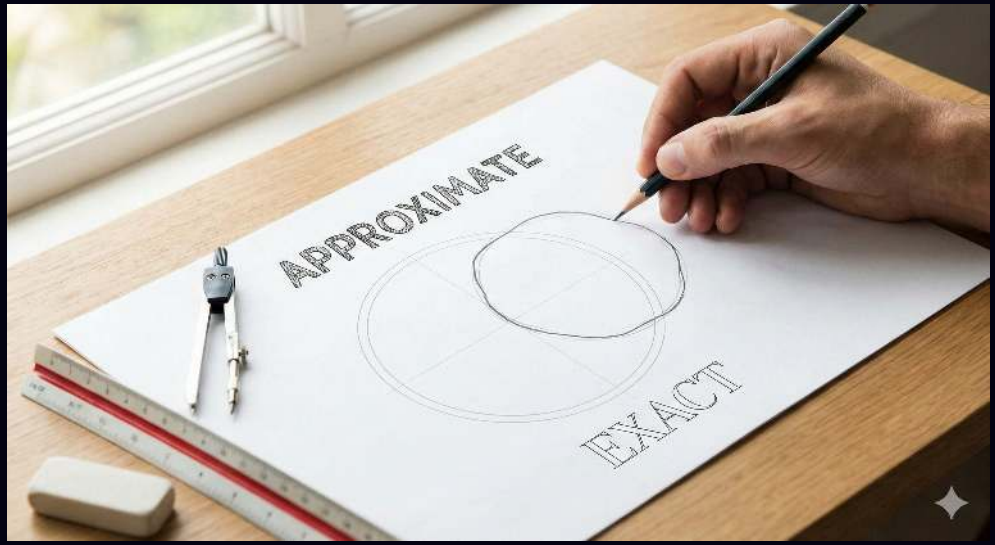
- Seq Scan on docs
- Reads all 50,000 rows
- ~500-2000ms query time
- High buffer reads (TOAST!)

Without indexes:

- Calculates distance for all 50k vectors
- Reads from TOAST for each
- O(N × dimensions) complexity

We need indexes! Like ANN (Approximate Nearest Neighbor)

Why Approximate? Because exact search is too slow!

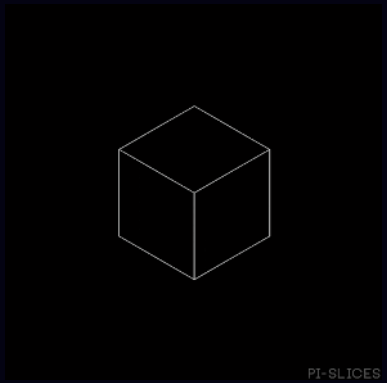


This is fine. 🔥🐶

Chapter 7: IVFFlat Index

Sequential scan is too slow. We need indexes!

But how do you index high-dimensional vectors?



Let's understand search methods first... apart from B+ tree variants used in for non vector indexing

IVFFlat - Inverted File Index with Flat Storage

The name tells the story:

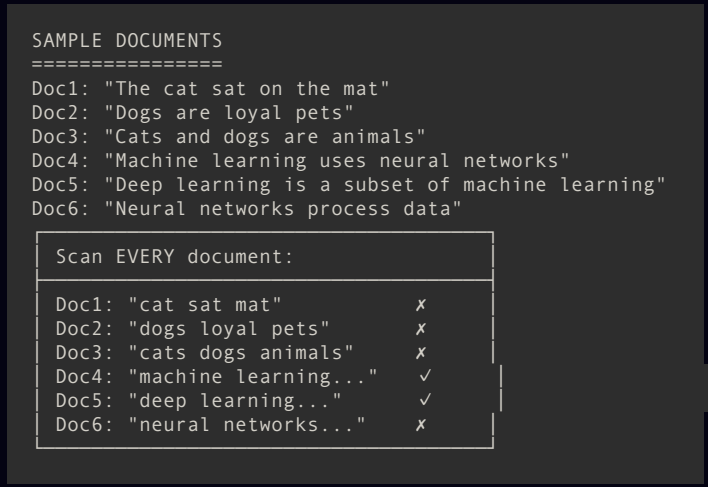
- Inverted File: Like a book index - points to where things are
- Flat: Vectors stored as-is

Let's see how different search methods work...

Sequential Scan (Brute Force)

Query: "machine learning algorithms"

To find semantic documents similar to this query



Checked: 6/6 docs (100%)

Problem: Must check EVERY document!

Inverted Index (TF-IDF)

Analogy from Elasticsearch's TF-IDF:

Pre-built index:

```
┌──────────────────────────────┐
│ "machine"  → [Doc4, Doc5]    │
│ "learning" → [Doc4, Doc5]    │
│ "neural"   → [Doc4, Doc6]    │
│ "cat"      → [Doc1, Doc3]    │
│ "dog"      → [Doc2, Doc3]    │
└──────────────────────────────┘
```

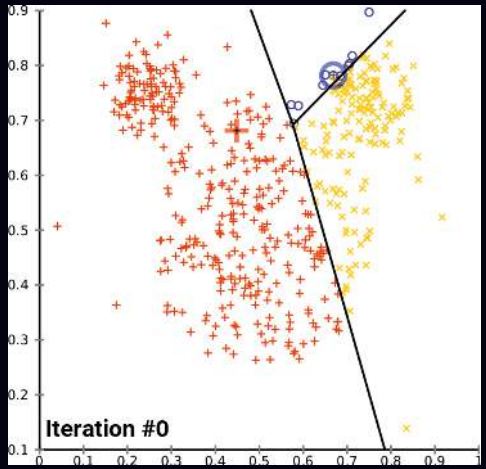
```
Query: "machine learning algorithms"
  ↓
Lookup: "machine" → [Doc4, Doc5]
Lookup: "learning" → [Doc4, Doc5]
Lookup: "algorithms" → []
  ↓
Intersect & rank: [Doc4, Doc5]
```

Checked: 2/6 docs (33%)

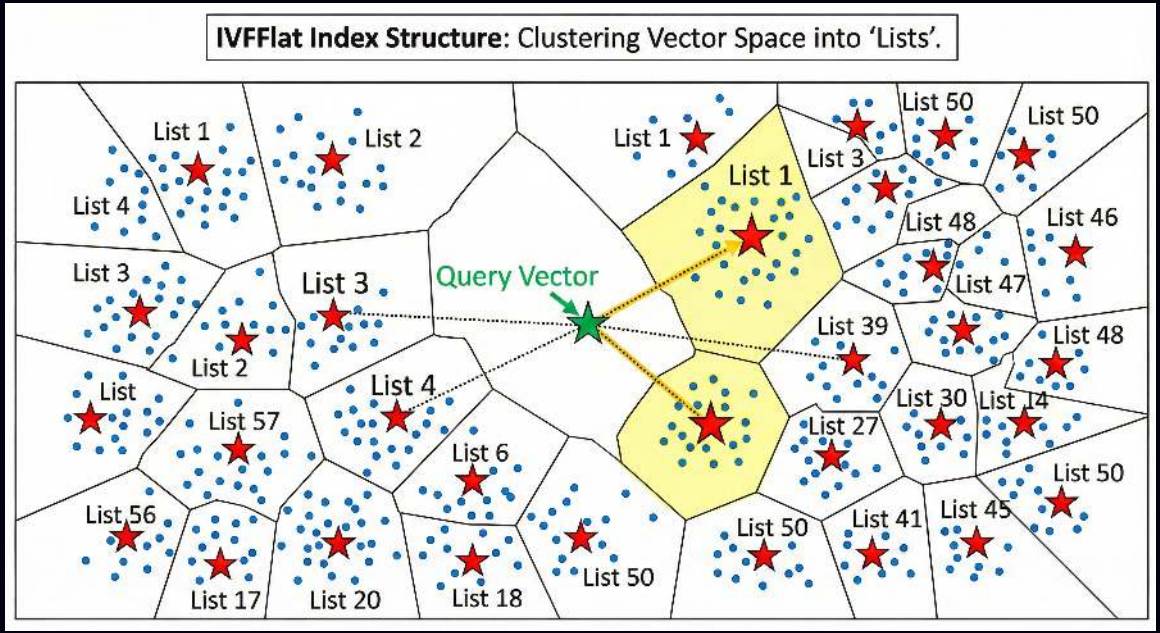
IVFFlat (Vector Embeddings)

How it works:

1. Documents converted to vectors
2. K-means clustering groups similar vectors
3. Query searches only nearest cluster(s)



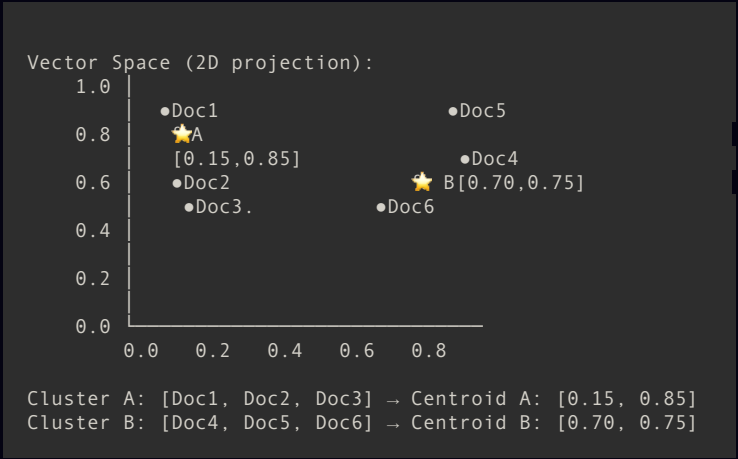
It's like organizing your closet by color. Sure, you might miss that one blue shirt, but you'll find A blue shirt fast!



Key insight: Don't search everything, just search the right neighborhood!

IVFFlat: Centroids Created

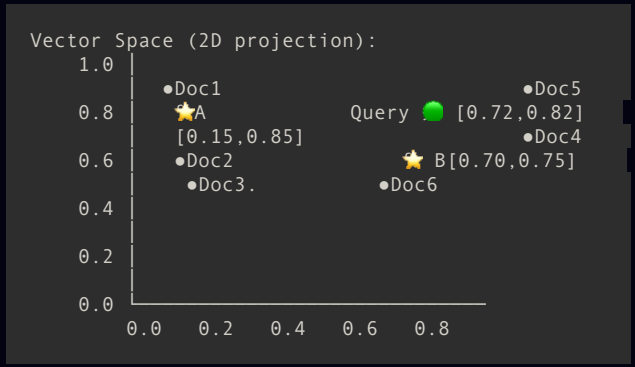
After k-means clustering, we have centroids (cluster centers):



Now let's see how search works...

IVFFlat Search Process

Query arrives: "machine learning algorithms" → [0.72, 0.82]

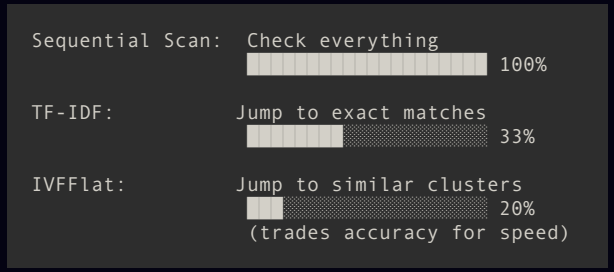


Which centroid is closest to the query?

IVFFlat Search Process: Step by Step

```
Query: "machine learning algorithms"
  → Embedding: [0.72, 0.82]  (2D projection)
  ↓
1. Find nearest centroid(s):
   Centroid A: [0.15, 0.25]
     Distance to A: 0.95  ✗

   Centroid B: [0.70, 0.80]
     Distance to B: 0.03  ✓ (nprobe=1)
  ↓
2. Scan ONLY Centroid B cluster:
   Doc4: distance 0.02  ✓
   Doc5: distance 0.04  ✓
   Doc6: distance 0.08  ✗
```

Search Methods Summary



Lets see IVFFlat Index 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;
```

Parameters:

- lists = 200 (≈ sqrt(50k) ≈ 224) - So 200 clusters
- vector_cosine_ops - cosine distance - `<=>` operator for normalized embeddings

```
SET enable_seqscan = off;
SET ivfflat.probes = 1;

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 using docs_ivfflat_idx
- ~10-20ms (vs 500-2000ms!)
- ~1,000 buffers (vs ~300,000!)
- 50-100x faster

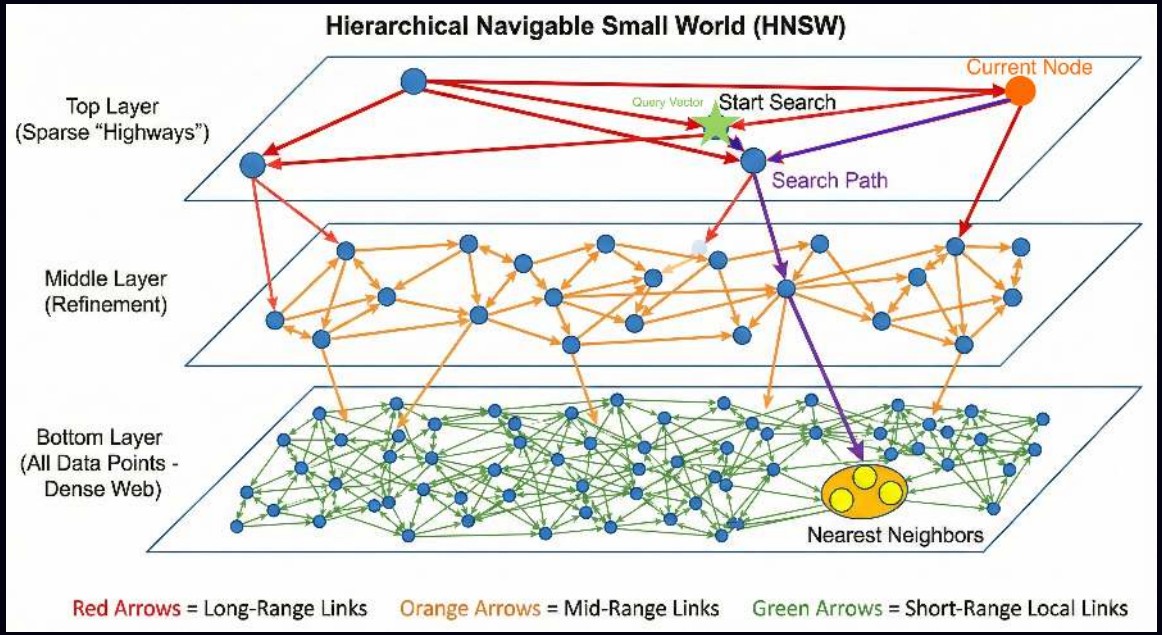
Chapter 8: HNSW Index

IVFFlat is fast, but...

- Can miss good results (lower recall)
- Depends on good clustering
- Needs tuning (probes parameter)

Can we do better? Enter HNSW!

Hierarchical Navigable Small World


Hierarchical Navigable Small World (HNSW)
Top Layer
(Sparse "Highways")
Middle Layer
(Refinement)
Bottom Layer
(All Data Points -
Dense Web)
Current Node
Query Vector
Start Search
Search Path
Nearest Neighbors
Red Arrows = Long-Range Links
Orange Arrows = Mid-Range Links
Green Arrows = Short-Range Local Links

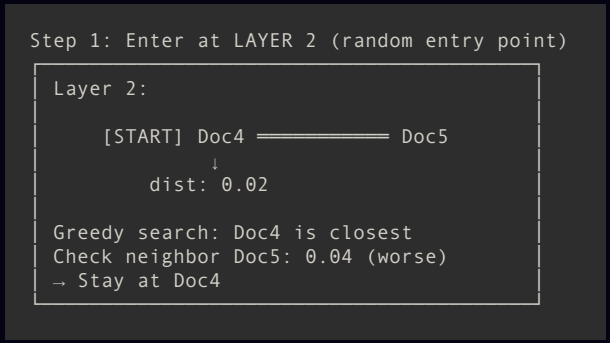
HNSW:

How it works:

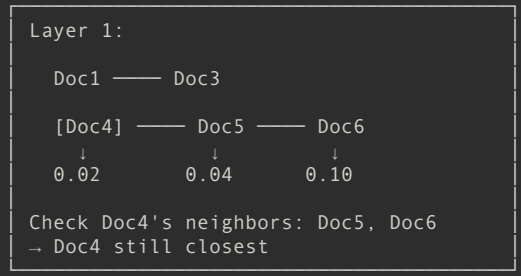
- Hierarchical graph with multiple layers
- Navigates from sparse (top) to dense (bottom) layers
- Higher recall
- More consistent performance

HNSW Search Process - Layer 2

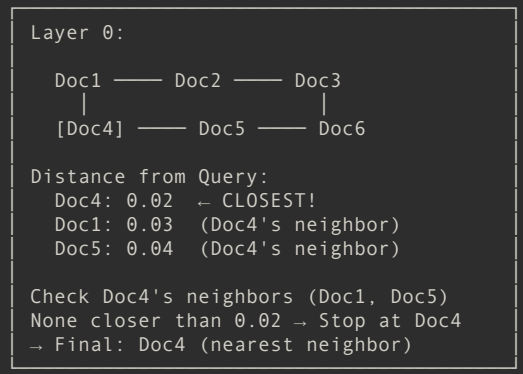
Query: "machine learning algorithms" → [0.72, 0.82, 0.08]



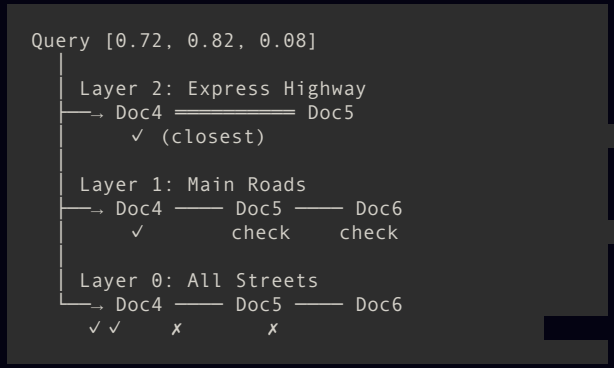
Step 2: Drop to LAYER 1 (from Doc4)



Step 3: Drop to LAYER 0 (from Doc4)



HNSW Path Visualization



Analogy: Like GPS navigation - use highways first, then local roads

Restaurant Analogy

Sequential Scan:

```
Walk every street, check every building
🏚 → 🏠 → 🏠 → 🏠 → 🏠 → 🏠 → 🍽

"Is this Italian? No.
 Is this Italian? No..."
Check ALL restaurants one by one
```

IVFFlat:

```
Jump to right neighborhood
   🏰 → [Downtown] → 🏠 → 🏠 → 🍽

Fly to Italian district
Search every restaurant in area
(even non-Italian)
```

TF-IDF:

```
Look up in phone book
📖 "Italian" → [Addr1, Addr2, Addr3]
🏎 → 🍽 (direct jump)

Only visit restaurants
labeled "Italian"
```

HNSW:

```
Highway → Avenue → Street
 🛣 → 🏎 → 🏚 → 🍽
 (L2)   (L1)   (L0)

Start at landmark restaurant
Follow signs to closer ones
```

🟦 Let's Try HNSW Index Now!

IVFFlat was good, but HNSW promises:

- ✅ Higher recall (~99% vs ~95%)
- ✅ More consistent performance
- ✅ Better for production workloads

Time to see it 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);
```

Parameters:

- `m = 16`: Max connections per node (higher = better recall, larger index)
- `ef_construction = 200`: Build-time search depth (higher = better quality, slower build)
- `vector_cosine_ops` : cosine distance - `<=>` operator for normalized embeddings

🟦 Does HNSW Deliver? Let's Find Out!

```sql
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 embedding <=> (SELECT embedding FROM docs WHERE id = 1)
LIMIT 5;
```

Parameter:

- `ef_search = 40`: Query-time search depth (higher = better recall, slower search)
  - Default: 40
  - Trade-off: Speed vs accuracy

Observe:

- ✅ Index Scan using docs_hnsw_idx
- ✅ Execution time: ~5-15ms

Performance Comparison

Method	Time	Speedup	Accuracy
Seq Scan	500-2000ms	1x	Perfect
IVFFlat	10-20ms	50-100x	Approximate
HNSW	5-15ms	70-150x	High

Key: 300x less I/O with indexes!

Build time: IVFFlat ~30-60s, HNSW ~2-5min

🟦 ⚠️ CRITICAL: Always Use LIMIT!

Without LIMIT, Postgres won't use ANN indexes!

```
-- BAD: No LIMIT = Sequential scan (even with indexes!)
SELECT * FROM docs
ORDER BY embedding <=> '[...]';

-- GOOD: LIMIT = Uses ANN index
SELECT * FROM docs
ORDER BY embedding <=> '[...]'
LIMIT 10;
```

Why? Without LIMIT, Postgres must sort ALL results → can't use approximate indexes.

> Classical queries scan and sort all rows to return the top N; vector search maintains a fixed-capacity leaderboard, dynamically swapping out weaker matches for better ones as it traverses the graph

Forgetting LIMIT is like asking for "all the fries" and wondering why it's slow 🍟

Chapter 9: Production Challenges

HNSW gives us great performance, but what about updates?

Let's talk about MVCC bloat...

MVCC Bloat - The Problem

What is MVCC? Multi-Version Concurrency Control - Postgres's way of handling concurrent transactions

How it works:

- UPDATE doesn't modify rows in-place
- Creates a NEW version, marks OLD version as dead
- Old versions stay until VACUUM cleans them up

Why it matters for vectors:

```
UPDATE docs SET metadata = '{"views": 100}' WHERE id = 1;
```

- Postgres copies the ENTIRE row (including 4KB embedding!)
- Old 4KB embedding becomes "dead tuple" in TOAST
- Do this 5000 times = 20 MB of dead data
- Scale to 50k docs with frequent updates = GBs of bloat!

Database is basically hoarding old versions like: "But I might need this someday!" 📦

Production Tips

Now that we understand the challenges, how do we build production-ready systems?

Let's go through 6 essential tips...

🟦 Production Tips #1

Separate embedding table

The Problem: Updating document metadata duplicates 4KB embeddings!

```
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
);

-- Update metadata only
UPDATE documents
SET metadata = '{"views": 100}'
WHERE id = 1;
```

✅ No embedding duplication ✅ Less TOAST bloat ✅ Re-embed only when content changes

Production Tips #2-3

We've separated tables. What about the embeddings themselves?

Use smaller models

- 384d = 1.5 KB (stays inline!)
- 512d = 2 KB (borderline)
- 1024d = 4 KB (TOASTed, our demo)

Smaller = less TOAST, faster queries

Increase maintenance_work_mem

```
-- In postgresql.conf
maintenance_work_mem = 512MB
```

Why: Vector indexes need memory to build (especially for 50k+ rows)

Production Tips #4-5

Smaller embeddings help, but what about indexing strategy?

Always ANALYZE after bulk inserts

```
ANALYZE docs;
```

Why: IVFFlat needs statistics for clustering

Create indexes AFTER bulk loading

```
-- Bad: index exists during inserts
CREATE INDEX FIRST;
INSERT lots of data; -- Slow!

-- Good:
INSERT lots of data; -- Fast!
CREATE INDEX AFTER; -- One-time cost
```

Why?

- IVFFlat: Incremental inserts can create poor clusters (local minima)
- HNSW: Each insert requires graph traversal and rebalancing (slow)

Best practice: Bulk load first, then index!

Production Tips #6

One more critical detail about vector queries...

Use LIMIT for ANN - It's Different!

Classical SQL:

```
-- LIMIT = "stop after N rows"
SELECT * FROM users
ORDER BY created_at
LIMIT 10;
```

1. Scan all rows
2. Sort all results
3. Return top 10

Vector Search:

```
-- LIMIT = "use ANN index!"
SELECT * FROM docs
ORDER BY embedding <=> '[...]'
LIMIT 10;
```

1. Think 10 like bucket
2. Add & Keeps the best 10 in that bucket
3. Discards others probabilistically

🟦 When to Use What?

Use IVFFlat when:

- ✅ Batch/analytics workloads
- ✅ Can tolerate ~95% recall
- ✅ Faster index build needed (~30-60s)
- ✅ Memory constrained (smaller index)
- ✅ Write-heavy workloads (faster inserts)

Example use cases:

- Nightly batch similarity jobs
- Data exploration/analysis
- Development/testing environments

Use HNSW when:

- ✅ Production user-facing apps (low-latency required)
- ✅ Need high recall (~99%)
- ✅ Read-heavy workloads (slow inserts acceptable)
- ✅ Can afford larger index size & build time

Example use cases:

- Real-time search APIs
- Recommendation systems
- Chatbot/RAG applications
- Production semantic search

TL;DR: HNSW for read-heavy production, IVFFlat for write-heavy or batch workloads

🟦 When to Use Alternatives in PostgreSQL

pgvector is great, but not always the answer:

Use PostgreSQL Full-Text Search (FTS) when:

- ✅ Exact keyword matching needed
- ✅ Boolean queries (AND, OR, NOT)
- ✅ Phrase matching required
- ✅ Language-specific stemming needed

Use sequential scan when:

- ✅ < 1,000 documents (brute force is fine)
- ✅ One-time queries (index build cost not worth it)
- ✅ 100% accuracy required (no approximation)

Use external vector DB when:

- ✅ > 10M vectors (specialized systems scale better)
- ✅ Need advanced features (filtering, hybrid search)
- ✅ Multi-tenancy with isolation

TL;DR: pgvector is perfect for 1k-10M vectors with PostgreSQL

🟦 Key Takeaways

The Big Picture:

PostgreSQL + pgvector handles 1k-10M vectors in production. No specialized database needed.

We covered:

- Storage Reality

- Index Strategy

- Production Gotchas

The Bottom Line:

ACID + SQL + existing infrastructure.

Postgres can do semantic search 🎉

The End

Postgres can handle semantic search at scale!

Now go forth and vector! 🚀

Questions?

📫 Get in touch: jeevan.dc24@alumni.iimb.ac.in
🌐 I write at noobj.me
(Not active on social media)

Slides & Code:

IVFFlat & HNSW Build Process Details

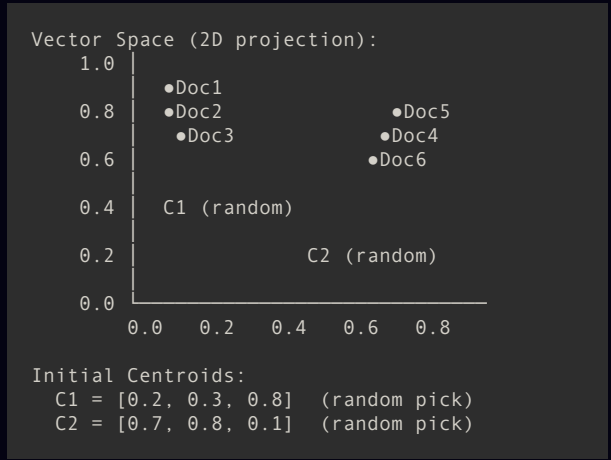
🟦 IVFFlat Step 0: Raw Data

```
Doc1: "The cat sat on the mat"        → [0.1, 0.2, 0.9]
Doc2: "Dogs are loyal pets"           → [0.2, 0.3, 0.8]
Doc3: "Cats and dogs are animals"     → [0.15, 0.25, 0.85]
Doc4: "Machine learning neural nets"  → [0.7, 0.8, 0.1]
Doc5: "Deep learning subset"          → [0.75, 0.85, 0.05]
Doc6: "Neural networks process data"  → [0.65, 0.75, 0.15]
```

⚠️ CRITICAL: Must load ALL vectors into memory first!

IVFFlat Step 1: Initialize Centroids

Randomly pick K=2 initial centroids:



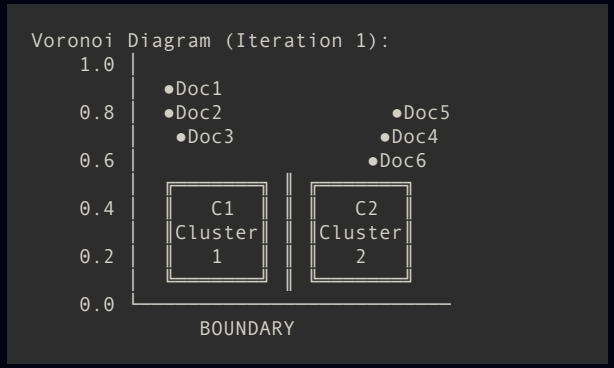
IVFFlat Step 2: Assign to Clusters (1/2)

Calculate distance from each vector to each centroid:

```
Doc1 [0.1, 0.2, 0.9]:
  → distance to C1: 0.14  ✓ (closer)
  → distance to C2: 0.95
  → Assign to Cluster 1

Doc2 [0.2, 0.3, 0.8]:
  → distance to C1: 0.00  ✓ (exact match!)
  → distance to C2: 0.85
  → Assign to Cluster 1

Doc3 [0.15, 0.25, 0.85]:
  → distance to C1: 0.08  ✓
  → distance to C2: 0.90
  → Assign to Cluster 1
```

```
Doc4 [0.7, 0.8, 0.1]:
  → distance to C1: 0.85
  → distance to C2: 0.00  ✓ (exact match!)
  → Assign to Cluster 2

Doc5 [0.75, 0.85, 0.05]:
  → distance to C1: 0.92
  → distance to C2: 0.08  ✓
  → Assign to Cluster 2

Doc6 [0.65, 0.75, 0.15]:
  → distance to C1: 0.78
  → distance to C2: 0.08  ✓
  → Assign to Cluster 2
```

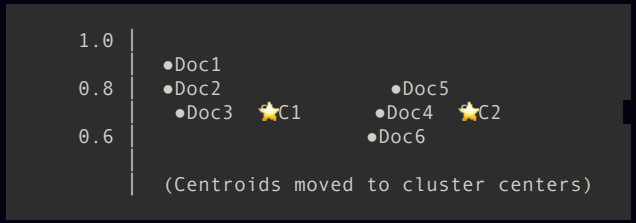
IVFFlat Voronoi Diagram



IVFFlat Step 3: Recalculate Centroids

```
Cluster 1: [Doc1, Doc2, Doc3]
  New C1 = mean([0.1, 0.2, 0.9],
                [0.2, 0.3, 0.8],
                [0.15, 0.25, 0.85])
  New C1 = [0.15, 0.25, 0.85]  ← MOVED!

Cluster 2: [Doc4, Doc5, Doc6]
  New C2 = mean([0.7, 0.8, 0.1],
                [0.75, 0.85, 0.05],
                [0.65, 0.75, 0.15])
  New C2 = [0.70, 0.80, 0.10]  ← MOVED!
```

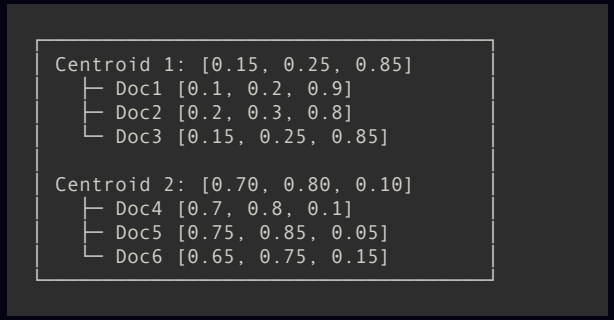
IVFFlat Centroids Moved



IVFFlat Step 4: Iterate Until Convergence

```
Repeat Steps 2-3 until centroids stop moving:

Iteration 2:
  - Recalculate distances
  - Reassign vectors
  - Update centroids

Iteration 3:
  - Recalculate distances
  - Reassign vectors
  - Update centroids

... CONVERGED! (centroids don't move)
```

IVFFlat Final Structure



HNSW Step 0: Start Empty

```
Doc1: "The cat sat on the mat"        → [0.1, 0.2, 0.9]
Doc2: "Dogs are loyal pets"           → [0.2, 0.3, 0.8]
Doc3: "Cats and dogs are animals"     → [0.15, 0.25, 0.85]
Doc4: "Machine learning neural nets"  → [0.7, 0.8, 0.1]
Doc5: "Deep learning subset"          → [0.75, 0.85, 0.05]
Doc6: "Neural networks process data"  → [0.65, 0.75, 0.15]
```

✓ NO need to load all data at once! ✓ Can insert one vector at a time

HNSW Parameters

```
M = 2        (max connections per layer)
efConstruction = 3  (search width during build)
mL = 1/ln(2) (layer probability multiplier)
```

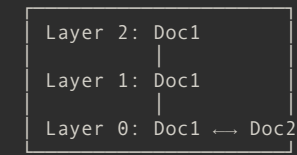
```
Step 1: Determine max layer
  Random: level = floor(-ln(random()) × mL) = 2

Step 2: Insert into all layers (0 to 2)

Layer 2:
  [Doc1]  ← First node, no connections

Layer 1:
  [Doc1]  ← First node, no connections

Layer 0:
  [Doc1]  ← First node, no connections

Current Index:
┌──────────────────┐
│ Layer 2: Doc1    │
│          │       │
│ Layer 1: Doc1    │
│          │       │
│ Layer 0: Doc1    │
└──────────────────┘
```

HNSW Insert Doc2

```
Step 1: Determine max layer
  Random: level = 0 (most nodes go to layer 0 only)

Step 2: Search for nearest neighbors
  Start at top layer with entry point Doc1

  Layer 2: Doc1 (entry point, skip - Doc2 not here)
  Layer 1: Doc1 (skip - Doc2 not here)
  Layer 0: Find nearest to Doc2
    → Doc1 distance: 0.14

Step 3: Connect Doc2 to M=2 nearest neighbors
  Layer 0: Doc2 ←→ Doc1
```

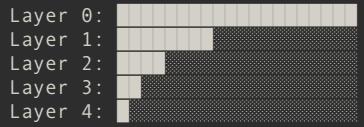
HNSW After Doc2

Current Index:



HNSW Layer Assignment

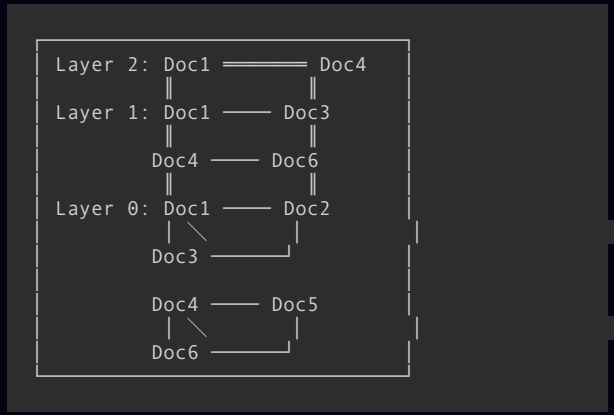
Each node gets random layer based on exponential decay:



Formula: P(layer ≥ L) = (1/2)^L

Creates a skip-list structure!

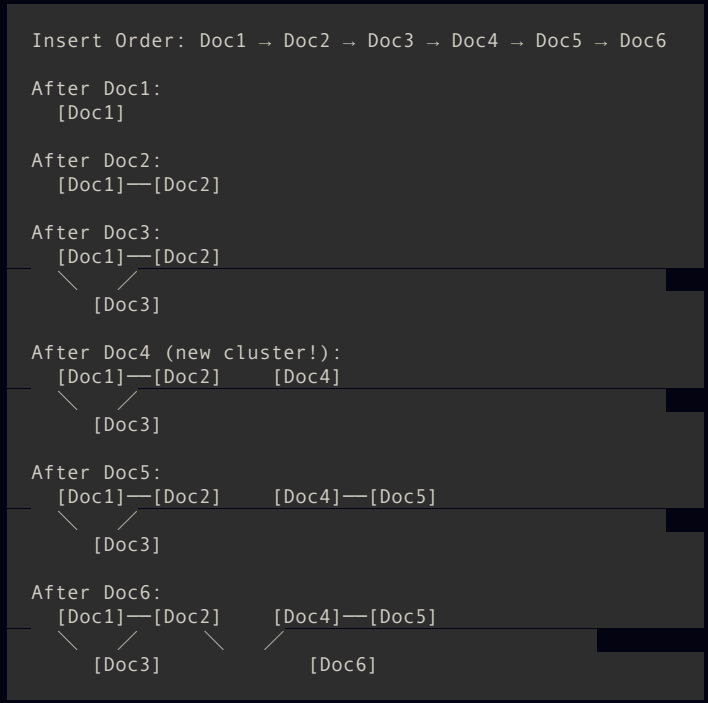
HNSW Final Structure



HNSW Connection Rules

During insertion:

1. Search Phase:
   - Start at top layer
   - Greedy search: move to closer neighbor
   - Drop layer when stuck
2. Connection Phase:
   - Find M nearest neighbors
   - Connect to them
   - Prune to maintain M max


Insert Order: Doc1 → Doc2 → Doc3 → Doc4 → Doc5 → Doc6
After Doc1:
[Doc1]
After Doc2:
[Doc1]──[Doc2]
After Doc3:
[Doc1]──[Doc2]
[Doc3]
After Doc4 (new cluster!):
[Doc1]──[Doc2] [Doc4]
[Doc3]
After Doc5:
[Doc1]──[Doc2] [Doc4]──[Doc5]
[Doc3]
After Doc6:
[Doc1]──[Doc2] [Doc4]──[Doc5]
[Doc3] [Doc6]

HNSW Tuning

```
-- Higher ef_search =
-- better recall, slower

SET hnsw.ef_search = 40;
-- Good default

SET hnsw.ef_search = 100;
-- Higher accuracy
```

Build parameters:

- `m = 16` max connections per node
- `ef_construction = 200` build quality

References

- HNSW: https://www.pinecone.io/learn/series/faiss/hnsw/
- HNSW: https://www.mongodb.com/resources/basics/hierarchical-navigable-small-world
- IVFFlat: https://milvus.io/docs/ivf-flat.md
- TOAST: https://www.postgresql.org/docs/current/storage-toast.html
- TOAST: https://www.tigerdata.com/blog/what-is-toast-and-why-it-isnt-enough-for-data-compression-in-postgres
- MVCC: https://www.postgresql.org/docs/7.1/mvcc.html