Proprietary + Confidential

Google Cloud

Scaling Semantic Search

Indexing and Searching at Scale with pgvector

7 March 2025

Proprietary + Confidential

Speakers

Eeshan Gupta
Software Engineer

Gunjan Juyal
Staff Software Engineer

Proprietary + Confidential

Agenda

CHALLENGES

- Quick recap of vectors - Domain, PG as Vector DB, and pgvector
- Scaling - pgvector limitations and challenges
- Tradeoffs: Recall-vs-QPS impact on memory and build-time
- Scaling challenge! <-> Usability problem?

SOLUTIONS

- Tuning instance and flags
- Techniques: Dimensionality reduction, Quantization
- Techniques: Filtered indexes, Iterative index scans
- Tuning: Tradeoffs, and a Useful Algorithm

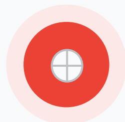
Recap - Why Vector DBs, and Why PG as a Vector DB?

pgvector is a very popular semantic "vector search" extension in PG. Let's see what that means.

Vector DB

Unlock the power of **Semantic Search..**

.. instead of exact, or finely-tuned brittle keyword or FTS search



PG as Vector DB

(instead of specialized ones)

Transactional **consistency**

No duplication of data between vector vs SQL DB

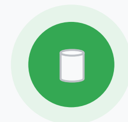
Architectural simplicity - lesser moving parts, less number of things that can go wrong

ANN Vector Search in PG

OSS:

1. pgvector (Most popular)
2. pgvecto.rs
3. pgvectorscale (adds an index type to pgvector)

Proprietary: ScaNN, DiskANN, etc



ANN indexes in pgvector

pgvector provides 2 index types:

1. IVFFlat
2. HNSW

HNSW vs IVFFlat

HNSW gives better performance than IVFFlat,

But is "expensive" (more on this in next slide)

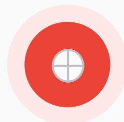
Examples - PG as a Vector DB

Semantic Search

Semantic Search = Search by "meaning" contained in the search query, rather than word or phrase match.

E.g. Image similarity search.

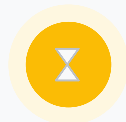
E.g. "Red food" would return "tuna fish" and "tomato gravy".



RAG

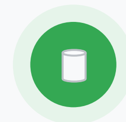
RAG or Retrieval Augmented Generation is a powerful new technique using vector search to improve Generative AI results.

E.g. Finding past similar bugs in PG, and using GenAI to compose an automated email for customer.



Recommendation system

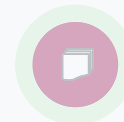
E.g. given a user's recent purchase history, return items most similar to them.



Multilingual Search

Search for articles like "climate change" in Spanish.

E.g. Vertex AI text-multilingual-embedding model family



Multimodal

Capture a photo and search for blog articles matching the contents of the image (or the other way around).

E.g. Vertex AI multimodalembedding model

Proprietary + Confidential

Recap - ANN vs KNN in Vector indexes

Why is vector index tuning hard? *(Psst.. They differ from traditional indexes such as B-Tree in profound ways)*

ANN vs KNN

ANN : Approximate Nearest Neighbours
KNN: K-Nearest Neighbours

- Vector is a data type
- Without a vector index, you would do an exhaustive "K-Nearest Neighbours" (KNN) search (brute-force)
- ANN-indexes provide an "approximate" answer that is aimed to be "good-enough", but at the fraction of cost

Recall-vs-QPS

Recall = An objective measure of "*good-enough*" that works for a particular use-case. E.g. 0.95 (=95%)

- Definition: If my exhaustive search for a given query-vector returns N neighbours, what % of these are returned by my ANN index?
- Recall and QPS - usually inverse relationship. Tradeoff!
- B-Tree: "Recall" is **Not** relevant, since "Recall" is always = 1. (Aside: But not for Bloom filters!)

Query Characteristics

Query shapes can greatly affect ANN performance. Unintuitive when coming from B-Tree world.

- **Top-K**: How many nearest neighbours need to be returned
- **Filtering**: Or the "WHERE" clause. Applied as "**post-filtering**", i.e. after a vector index has returned some nearest neighbours
- **Pgvector limit** on top-K = **1000**. And this is even before any SQL "WHERE" clauses are applied!

Proprietary + Confidential

Recap - ANN vs KNN in Vector indexes

Embeddingsssss

Why is vector index tuning hard? *(Psst.. They differ from traditional indexes such as B-Tree in profound ways)*

What index type?
IVFFlat, HNSW, ScaNN

Quantization? Type:
Halfvec
Bit

ANN vs KNN

ANN : Approximate Nearest Neighbours
KNN: K-Nearest Neighbours

Vector space - Sparse or not?

- Vector is a data type
- Without a vector index, you would do an exhaustive "K-Nearest Neighbours" (KNN) search (brute-force)
- ANN indexes provide an "approximate" answer that is aimed to be "good-enough" but at the fraction of cost

ivfflat.probes

Ef_construction,
ef_search

Recall-vs-QPS

Recall = An objective measure of "*good-enough*" that works for a particular use-case. E.g. 0.95 (=95%)

- Definition: If my exhaustive search for a given query-vector returns N neighbours, what % of these are returned by my ANN index?
- Recall and QPS - usually inverse relationship. Tradeoff!
- B-Tree: "Recall" is **Not** relevant, since "Recall" is always = 1. (Aside: But not for Bloom filters!)

Recall/Performance curve

Hybrid Search
Filtering + ANN

Query Characteristics

DB Flags for tuning index build

Query shapes can greatly affect ANN performance. Unintuitive when coming from B-Tree world.

- **Top-K:** How many nearest neighbours need to be returned
- **Filtering:** Or the "WHERE" clause. Applied as "**post-filtering**", i.e. after a vector index has returned some nearest neighbours
- **Pgvector limit on top-K = 1000.** And this is even before any SQL "WHERE" clauses are applied!

Follow the white rabbit

The Challenge

Testimonial - pgvector in Production

It's awesome! I love it! *(But sometimes I think that life doesn't need to be so complex, no?)*

Proprietary + Confidential

Challenge of scaling in pgvector indexes

Pgvector indexes are hard to scale.. Especially HNSW.

Bottlenecks everywhere

Pgvector indexes and queries can hit several bottlenecks jeopardizing perf

- Incorrectly sized instance
- Misconfigured database params
  - `maintenance_work_mem`
  - `max_parallel_maintenance_workers`
  - `shared_buffers`
- Suboptimal index params
  - `M, ef_construction`

Proprietary + Confidential

Challenge of scaling in pgvector indexes

Pgvector indexes are hard to scale.. Especially HNSW.

Current scale

This is the scale that pgvector is able to handle, if configured correctly

- Number of vectors supported:
  - Millions - yes
  - Billions - Maybe (Hard)
  - 10s Billions - No!

Scaling challenges

Pgvector indexes face the following scaling challenges

- QPS <-> Recall
- Time to build index
- Memory
- Workloads - Read vs Write heavy
- Filtering - High-selectivity queries are a problem for ANN + WHERE clause queries)

Limitations

Pgvector comes with certain inherent limitations as well

- "Vector" type only supports upto 2000 dimensions
- "Top-K" has a max-limit of 1000
- "Top-K" becomes much lower in case of post-filtering (additional WHERE clauses)

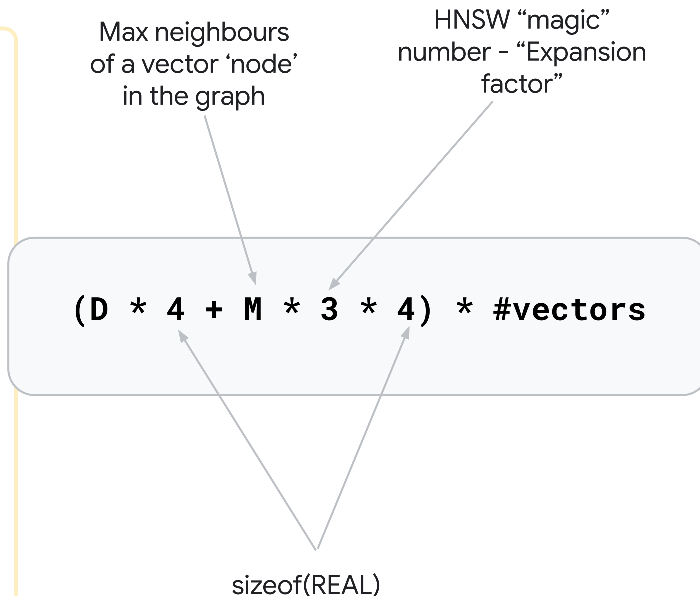
Proprietary + Confidential

Challenge of scaling in pgvector indexes

Pgvector indexes are hard to scale.. Especially HNSW.

Consider memory usage for HNSW

- Per vector~: `d * 4 + M * 3 * 4` Bytes
- Total memory: `(D * 4 + M * 3 * 4) * #vectors` Bytes
- Example for 1M vectors: M=32, dims=1536 will consume ~((1536 * 4) + (32 * 3 * 4)) * 1,000,000 = ~6.5 GB
- Same example with 10M vectors: 65 GB!
- 100M vectors: 650 GB!!!



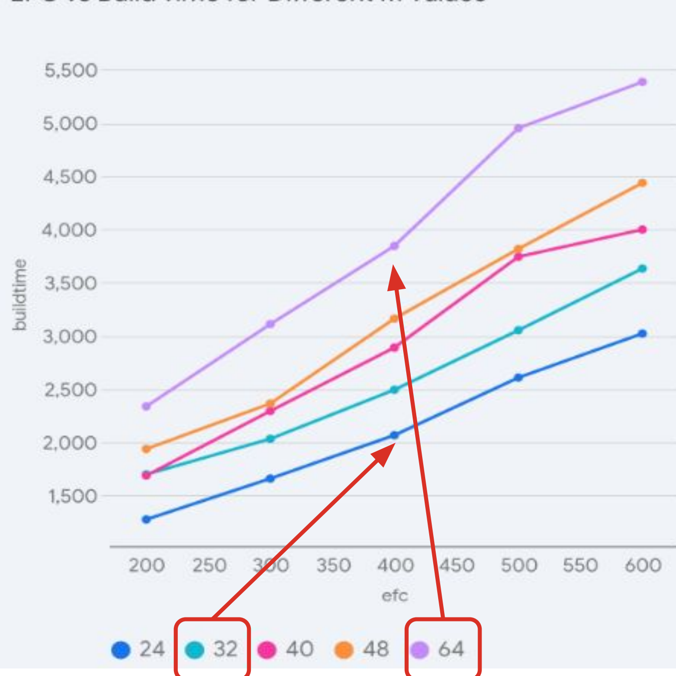
Proprietary + Confidential

Challenge of scaling in pgvector indexes

Pgvector indexes are hard to scale.. Especially HNSW.

Consider index build times for HNSW - some examples with M = 24, ef_construction = 200, dims = 128

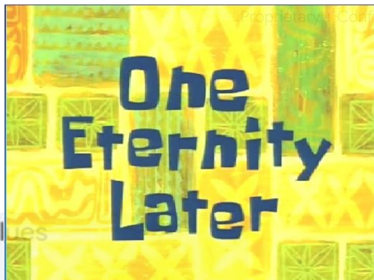
- 1M vectors, build time = 355 sec
- 10M vectors, build time = 3,688 sec (~1h!)
- 100M vectors, build time = 63,602 sec (~18h!!!)



Proprietary + Confidential

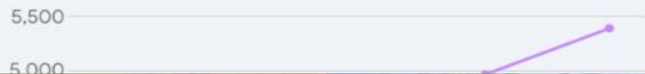
Challenge of scaling in pgvector indexes

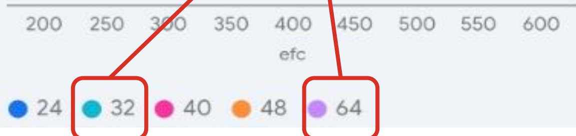
Pgvector indexes are hard to scale.. Especially HNSW.



Consider index build times for HNSW - some examples with M = 24, ef_construction = 200, dims = 128

- 1M vectors, build time = 355 sec
- 10M vectors, build time = 3688 sec (~1h!)
- 100M vectors, build time = 63602 sec (~18h!!!)





Proprietary + Confidential

Challenge of scaling in pgvector indexes

Pgvector indexes are hard to scale.. Especially HNSW.

Query Filtering - Pre vs Post:

- Pre-filtering applies the filter first and then performs brute-force search. It does not support vector indexes at all - No scalability

- ANN index can only be used in “Post-filtering” - Perform ANN search and then apply the filter. This can lead to fewer results than expected

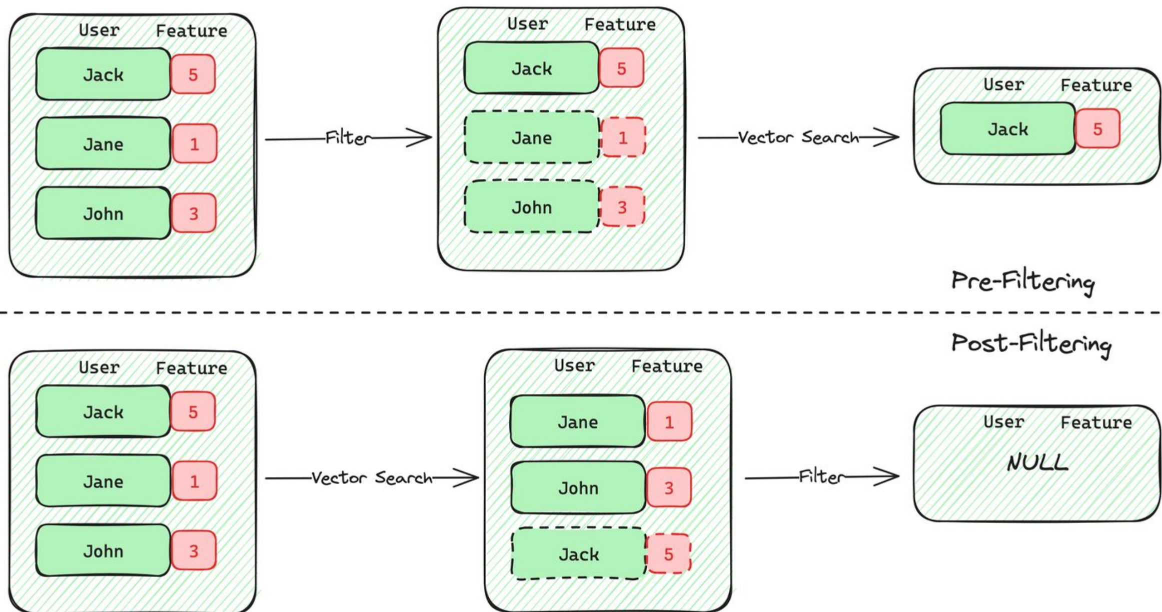


Image ref: https://myscale.com/blog/filtered-vector-search-in-myscale/

Proprietary + Confidential

Scaling Challenge is (sometimes) due to a Usability Problem

*“**It works when done correctly!** (But took me a PhD to reach that stage)”*

Proprietary + Confidential

Scaling Challenge is (sometimes) due to a Usability Problem

*“**It works when done correctly!** (But took me a PhD to reach that stage. And lots of socks with holes!)"*

From Challenge

To Solution

Proprietary + Confidential

Tuning your PostgreSQL DB for maximum scale

Flags Flags Everywhere! But you can start with just these ones.

Maximize Available Memory

This is the scale that pgvector is able to handle, if configured correctly

- `maintenance_work_mem`
- `shared_buffers`

Parallelize

Pgvector indexes face the following scaling challenges

- `max_parallel_maintenance_workers`

Example for a typical machine:

- Machine type: 8 vCPUs
- `max_parallel_maintenance_workers = 7 or 8`

Proprietary + Confidential

Dimensionality Reduction

Proprietary

Proprietary + Confidential

Dimensionality Reduction

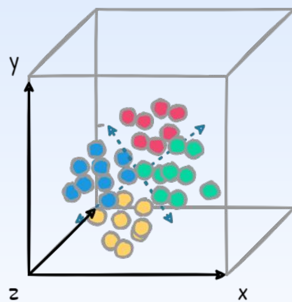
#Shift-Left: "Compress" your vectors - Less dimensions **is more** vectors in memory!

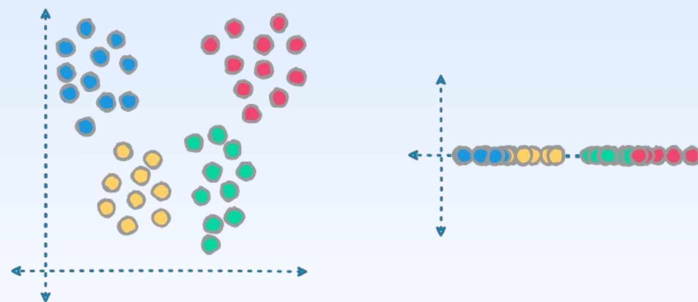
Reduce #dimensions if it doesn't lead to much information loss

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)

Dimensionality Reduction





Quantization

Proprietary + Confidential

Quantization for scaling

Do more with less. *(No free lunches in life, though.)*

What is Quantization

Essentially, Reducing vector precision
E.g. wav -> mp3

Think of it as compressing your vectors, trading a tiny bit of accuracy for much smaller index size and faster distance calculations

Why it helps with scale?

Pgvector indexes face the following scaling challenges

Smaller index = faster index building, faster queries, less memory. Highlight benefits for large datasets and high-dimensional vectors. (~1 minute)

We're essentially grouping similar vectors together in a lower-dimensional space

Types of Quantization

Pgvector supports 2 types of quantization for different degrees of accuracy-compression tradeoffs

- Halfvec
- Bit

Other promising quantization techniques not yet supported in pgvector:

- Product Quantization
- SQ8 (half of halfvec)

Proprietary + Confidential

Quantization Example

DDL - Regular

```
CREATE INDEX ON items USING hnsw (embedding vector_l2_ops) WITH (m = 16,
ef_construction = 64);
```

DDL - Quantized

```
CREATE INDEX ON items USING hnsw ((embedding::halfvec(768))
halfvec_l2_ops) WITH (m = 16, ef_construction = 64);
```

Proprietary + Confidential

Quantization Example

DQL - Regular:

```
SELECT * FROM products ORDER BY description_embeddings <=>
embedding(‘text-embedding-005’, ‘Help me scale pgvector!’) LIMIT 20;
```

DQL - Quantized:

```
SELECT * FROM products ORDER BY description_embeddings::halfvec(768) <=>
embedding(‘text-embedding-005’, ‘Help me scale pgvector!’) LIMIT 20;
```

DQL - Quantized with Re-ranking:

```
SELECT * FROM (

    SELECT * FROM items ORDER BY binary_quantize(embedding)::bit(3) <~>
binary_quantize('[1,-2,3]') LIMIT 20

) ORDER BY embedding <=> '[1,-2,3]' LIMIT 5;  -- re-ranking
```

Proprietary + Confidential

Quantization Perf benefits: *Vector vs Halfvec*

Dataset: *dbpedia-openai-1000k-angular*

Index: *HNSW* **M**: *16*

Using halfvec instead of vector - Index **build**-time.

ef_construction	Index size reduction	Build time Speedup
32	2.00x	3.17x
64	2.00x	2.93x
128	2.00x	2.62x

Using halfvec instead of vector - Index **search**-time.

hnsw.ef_search	vector recall	halfvec recall	vector QPS	halfvec QPS
40	96.8%	96.8%	567	578
200	99.6%	99.6%	156	163
800	99.9%	99.9%	48	51

Vector: ***??*** *byte floats* **Halfvec**: ***??*** *byte floats*

Ref: https://jkatz05.com/post/postgres/pgvector-scalar-binary-quantization/

Proprietary + Confidential

Quantization Perf benefits: *Vector vs Halfvec*

Dataset: *dbpedia-openai-1000k-angular*

Index: *HNSW* **M**: *16*

Using halfvec instead of vector - Index **build**-time.

ef_construction	Index size reduction	Build time Speedup
32	2.00x	3.17x
64	2.00x	2.93x
128	2.00x	2.62x

Using halfvec instead of vector - Index **search**-time.

hnsw.ef_search	vector recall	halfvec recall	vector QPS	halfvec QPS
40	96.8%	96.8%	567	578
200	99.6%	99.6%	156	163
800	99.9%	99.9%	48	51

Vector: *4 byte floats* **Halfvec**: *2 byte floats*

Ref: https://jkatz05.com/post/postgres/pgvector-scalar-binary-quantization/

Filtering Techniques

Proprietary + Confidential

Walk around the challenge : Pre-filtering indexes

In specific scenarios where that may be a feasible option

Create an index on the filtered column, then do exact-search

A good place to start is creating an index on the filter column. This can provide fast, exact nearest neighbor search in many cases.

However after filtering, the ANN index can no longer be used. This severely limits the scalability unless the selectivity of the filter is very high leading to feasible exact searches.

```
Query
SELECT * FROM items WHERE category_id = 123
ORDER BY embedding <-> '[3,1,2]' LIMIT 5;

Pre-filtering Indexes

CREATE INDEX ON items (category_id);

CREATE INDEX ON items (location_id,
category_id); -- multiple columns
```

Proprietary + Confidential

Partial ANN Indexes

For low-cardinality columns create separate index per "group"

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 category_id = 123
ORDER BY embedding <-> '[3,1,2]' LIMIT 5;
```

Partial ANN Index

```
CREATE INDEX ON items USING hnsw (embedding
vector_l2_ops) WHERE (category_id = 123);
```

Proprietary + Confidential

Partitioned ANN Indexes

Create partitioned tables and indexes on each partition

Use partitioned table with ANN indexes on the partitioned table

Partitioning is a powerful technique in PostgreSQL and can be leveraged to scale vector search as well

```
Query
SELECT * FROM items WHERE category_id = 123
ORDER BY embedding <-> '[3,1,2]' LIMIT 5;

Partitioned Tables and Indexes

CREATE TABLE items (embedding vector(3),
category_id int) PARTITION BY LIST(category_id);


CREATE INDEX ON items USING hnsw (embedding
vector_l2_ops)
```

Proprietary + Confidential

Use Iterative Index Scans

Pgvector 0.8.0 onwards supports 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

This will automatically scan more of the index until enough results are found.

It is done iteratively, as opposed to the traditional single loop.

Cost - Higher latency, more memory usage

```
Iterative Index Scan

SET hnsw.iterative_scan = strict_order;

SET hnsw.iterative_scan = relaxed_order;
```

Tuning Techniques

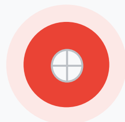
Proprietary + Confidential

Recap - Choosing the right Index Type

HNSW vs IVFFlat. TL;DR - HNSW is more performance but expensive to build/serve

Query Speed

HNSW is faster



Recall

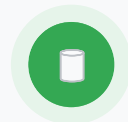
HNSW typically gives higher recall for same QPS

Caution: Be cognizant of data and query characteristics that can affect recall, e.g.:

* Query Selectivity
* Sparse vs Dense vectors

Index Building

IVFFlat is faster



Memory Usage

IVFFlat is lower

Incremental Data Changes

HNSW handles these well

IVFFlat is more sensitive, and may need more frequent re-building

Ref: pgvector docs - https://github.com/pgvector/pgvector/pull/422/files , Google Cloud blog on getting started with pgvector indexes.
Note: We'll focus on HNSW for most of the Tuning section

Proprietary + Confidential

Recap - Make conscious Tradeoffs in ANN indexes

Oh, it's always about "Tradeoffs".. Especially in ANN indexes and search!

FR and NFRs (of ANN indexes):

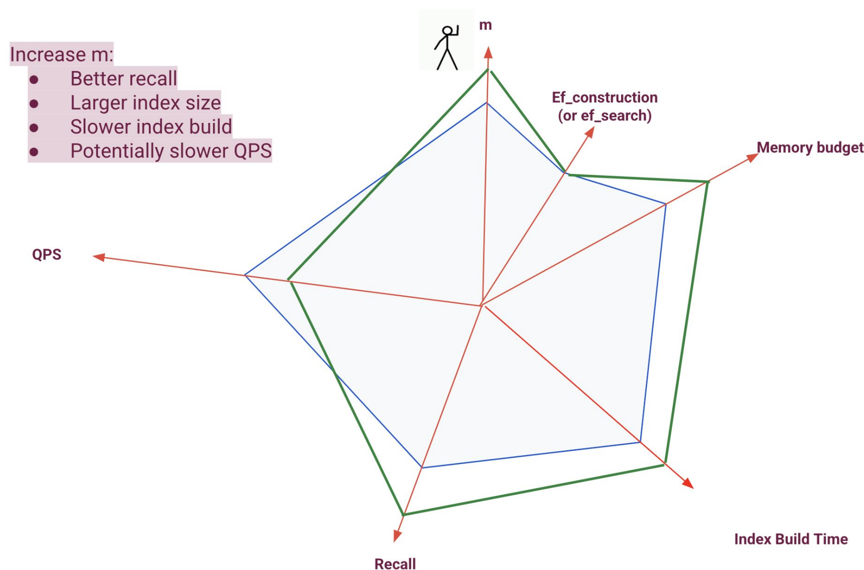
- Recall (Not a concern for exact indexes such as B-Tree)
- QPS, Latency
- Time to Build index

Resources / Constraints:

- Memory: Index time, Search time
- CPU: #Cores, Hardware

Index Config Parameters (HNSW):

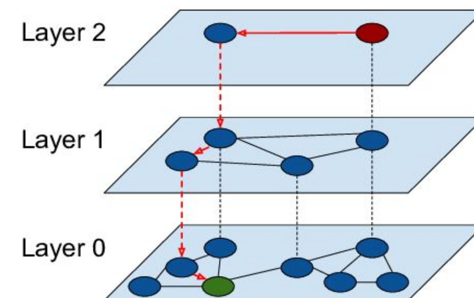
- M, ef_construction, ef_search
- Top-K?
- Selectivity, or query shape?



Choose the right index type.. then Tune index parameters

Parameters are index-type specific, we'll focus on HNSW here

Parameter	Applicable during	Impact on Increasing the parameter value	
		Benefit	**Cost**
m	Index Build	Give higher recall and/or Support higher dimensionality	Increased memory usage and Query time
ef_construction	Index Build	Improved recall (up to an extent)	Increased index build time
ef_search	Search Query	Higher recall	Lower QPS



Proprietary + Confidential

Choose the right index type.. then Tune index parameters

Combinatorial Explosion!!

Especially when also including Recall-vs-QPS tradeoff computation

Parameter	Applicable during	Impact on Increasing the parameter value	
		Benefit	Cost
m	Index Build	Give higher recall and/or Support higher dimensionality	Increased memory usage and Query time
ef_construction	Index Build	Improved recall (up to an extent)	Increased index build time
ef_search	Search Query	Higher recall	Lower QPS

Proprietary + Confidential

Choose the right index type.. then Tune index parameters

Grid-Search Method - Determining best parameters for HNSW. (*Exciting stuff! For some folks at least. I guess🤷🏻‍♂️*)

Typical values - for reference

- `M`: [8, 16, 24, 32, 48, 64]
- `ef_construction`: [64, 128, 256, 512]
- `Target recall =` 0.95

How to tune:

- Grid search for hyperparam tuning of all <M, ef_construction> pairs
- Evaluate QPS at target recall. Keep track of index build time and memory.

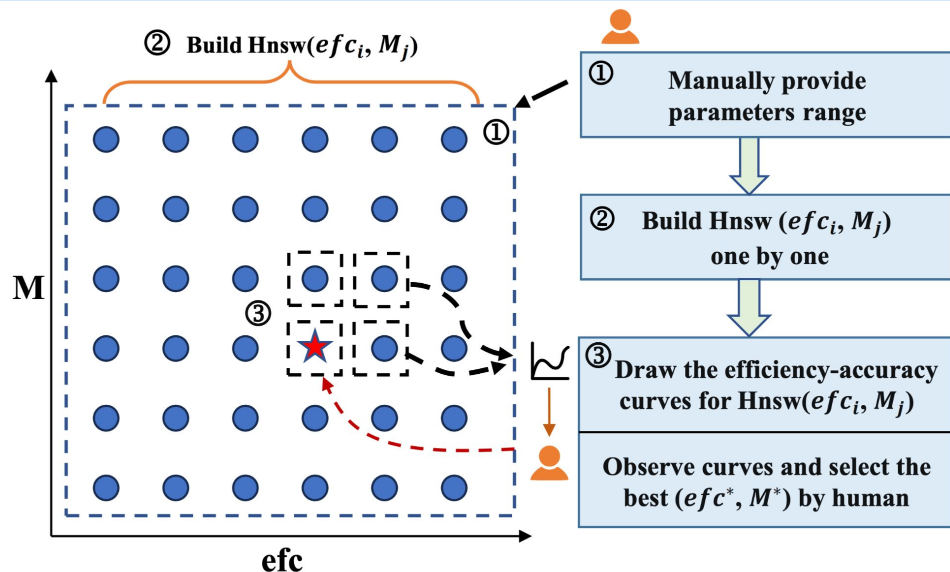


Image credits: HNSW Construction Params Auto Tuning paper by Zhou et al - https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4925468

Note: Many other algorithms and heuristics are possible for reducing the search space. Exciting stuff! (For some folks at least. I guess.)

Choose the right index type.. then Tune index parameters

Grid-Search Method - Determining best parameters for HNSW. (*Exciting stuff! For some folks at least. I guess🤷🏻‍♂️*)

How to tune - FASTER!:

- Eliminate <M, ef_construction> pairs using heuristics: For a M if higher efc gives diminishing returns then move to next M
- Stop when higher M gives lower QPS
- *More sophisticated heuristics exist - reach out if interested!*

Sampling:

Evaluate grid-search on a sample (100k or 10% whichever is lower) - results might be "close-enough" to running on full dataset

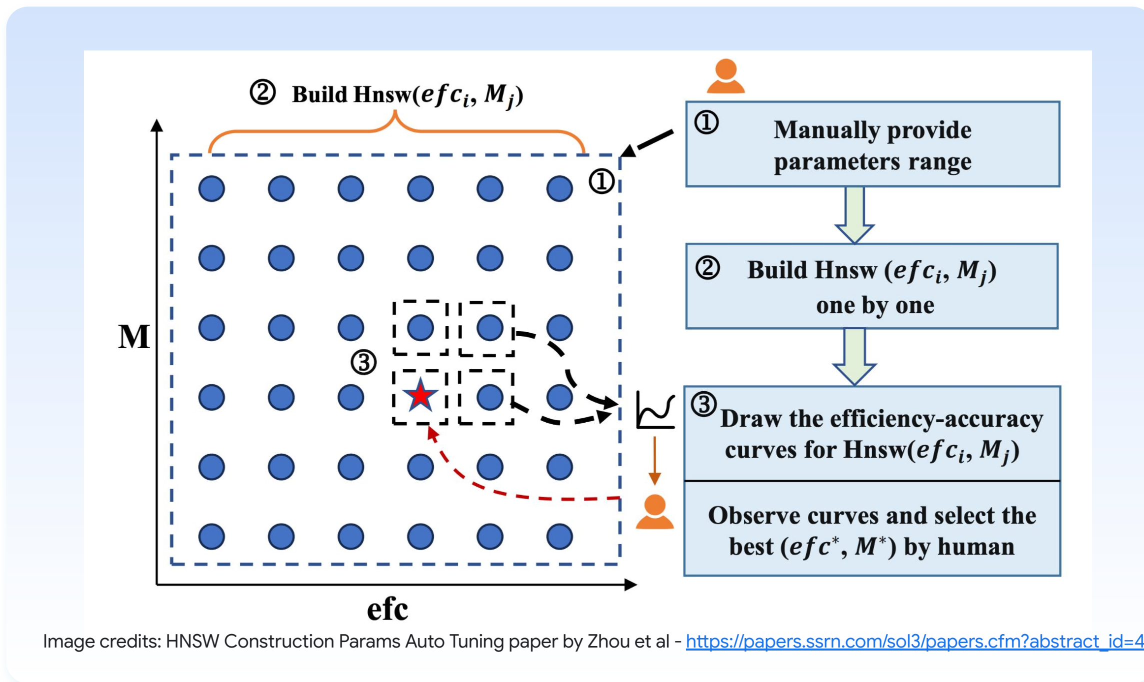


Image credits: HNSW Construction Params Auto Tuning paper by Zhou et al - https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4925468

Note: Many other algorithms and heuristics are possible for reducing the search space. Exciting stuff! (For some folks at least. I guess.)

Proprietary + Confidential

Choose the right index type.. then **Tune** index parameters

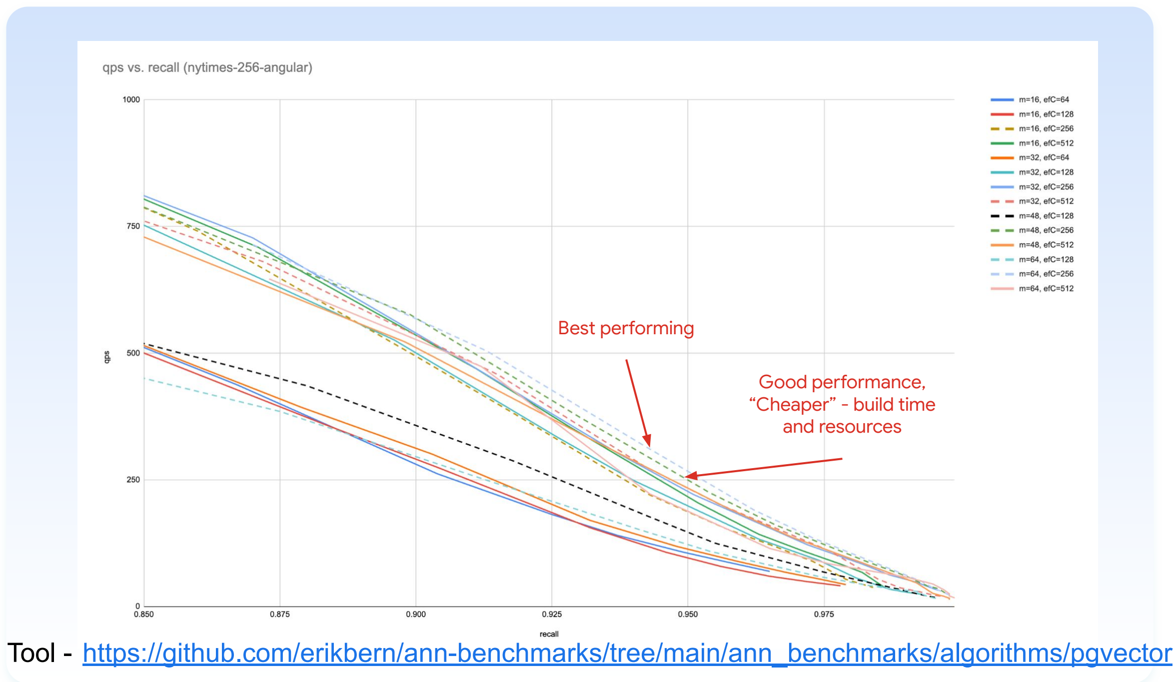
Grid-Search Method - Determining best parameters for HNSW. *(Exciting stuff! For some folks at least. I guess🤷🏻‍♂️)*

Recall:

- Computed for a randomly selected query from dataset
- Find the minimum ef_search that gives the given recall

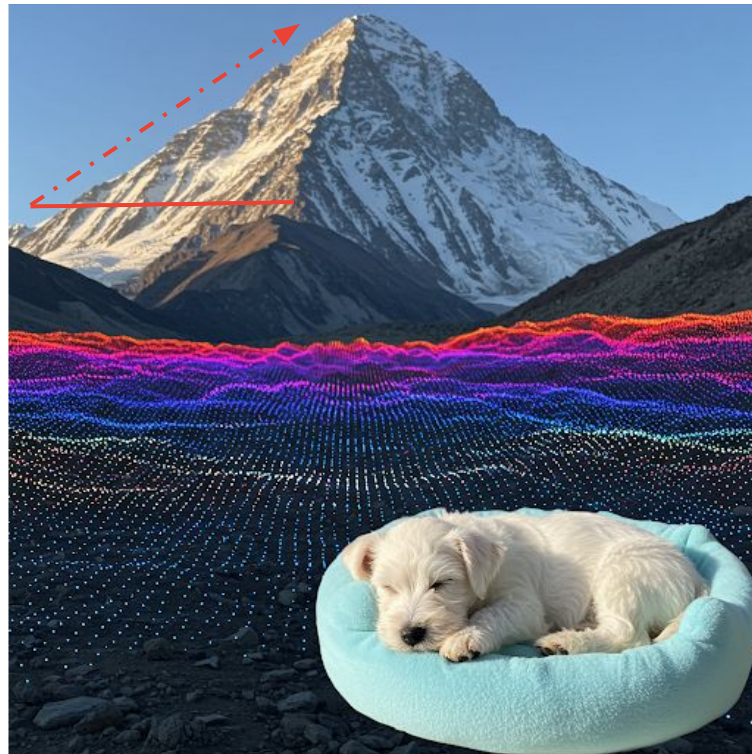
Choosing the “best” curve:

- Best performing curve is the one that gives highest QPS at target recall
- However, there might be curves giving “close-enough” QPS at lower cost (build time or memory)



Tool - https://github.com/erikbern/ann-benchmarks/tree/main/ann_benchmarks/algorithms/pgvector

Note: Many other algorithms and heuristics are possible for reducing the search space.



May your pgvector scale great heights!

Proprietary + Confidential

Thank You!

Eeshan Gupta
Software Engineer at Google

Gunjan Juyal
Google CloudSQL for PostgreSQL |
Software Engineer, data at scale | Google,...

Eeshan Gupta
Software Engineer

LinkedIn: @eeshangupta

Gunjan Juyal
Staff Software Engineer

LinkedIn: @gunjanjuyal