EDB POSTGRES AI

High-Throughput Vector Search at Scale:
pgvector HNSW vs. VectorChord Performance Test Report

enterprisedb.com

Contents

1. Executive summary

Enterprise organizations are increasingly deploying autonomous agents that depend on fast, accurate vector search at the heart of their data infrastructure. As those workloads scale—more vectors, more concurrent queries, larger embedding models—the choice of vector index becomes a primary determinant of performance. EDB Postgres® AI is built to support this generation of agentic applications natively, combining transactional rigor with multi-model capabilities including vector search. This report benchmarks two of the most capable PostgreSQL vector extensions—pgvector (HNSW index) and VectorChord (IVF-RaBitQ index)—to provide a clear, data-driven basis for understanding which index type serves which workload, and why the ability to deploy either within a single platform matters for enterprise agent infrastructure.

This report presents a comprehensive performance comparison between **pgvector** (HNSW index) and **VectorChord** (IVF-RaBitQ index) for approximate nearest neighbor (ANN) search in PostgreSQL. We evaluate both extensions across three dataset scales—5 million, 100 million, and 1 billion vectors[1]—measuring query throughput (QPS), recall, latency, index build time, concurrency scaling, and sensitivity to server memory configuration. Recall, the fraction of true nearest neighbors correctly returned by an approximate search, is the primary accuracy metric for ANN systems. Both engines were swept across their search parameters and benchmarked at the operating points nearest ~0.95 recall; throughput and latency comparisons in this report reflect architectural efficiency at comparable accuracy.

On our test hardware, VectorChord's IVF-RaBitQ index delivers 1.5x–2.4x higher query throughput than pgvector's HNSW index at equivalent memory, and it builds indexes 5x–12x faster across dataset sizes from 5M to 1B vectors. VectorChord's throughput also stays within 10% of its peak across a range of database memory allocations (16 GB to 512 GB), whereas pgvector HNSW's throughput varies by 5x–7x over the same range.

Key findings:

- **At small scale (5M vectors)**, VectorChord delivers **33% higher single-client QPS** and **45% higher concurrent QPS** (32 clients) than pgvector at equivalent recall (~0.95), with **5x faster index builds** and lower memory requirements.
- **At medium scale (100M vectors)**, VectorChord achieves **2x the single-client QPS** and **2.4x the concurrent QPS** of pgvector at equivalent recall. When `shared_buffers` is constrained to 16 GB and 64 GB total available RAM on our test hardware, the gap widens to 14x single-client and 12x concurrent—pgvector HNSW's random I/O pattern saturates storage while VectorChord's sequential scans remain efficient.
- **At large scale (1B vectors)**, VectorChord builds the index **13x faster** (1.2h vs. 15.7h), produces a **25% smaller index** (482 GB vs. 646 GB), and under concurrent load delivers **1,267 QPS vs. 818 QPS**—a **55% advantage**—with half the tail latency (P99 35.7 ms vs. 66.9 ms).
- **VectorChord is nearly immune to server memory sizing.** QPS varies less than 10% across `shared_buffers` sizes from 16 GB to 512 GB. pgvector HNSW is highly sensitive to memory sizing and suffers a double-buffering penalty at intermediate sizes.
- **Under concurrency, pgvector HNSW becomes I/O-bound.** With 32 clients on a 1B index at 32 GB `shared_buffers`, all pgvector backends block on `BufferIo` waits—CPUs sit idle while storage is saturated with random reads. VectorChord scales nearly linearly, achieving **31x throughput** from 32 clients under the same conditions.
- **The advantage compounds with scale and load.** VectorChord's performance benefits widen as datasets grow and concurrency increases—the architectural difference between sequential and random I/O becomes more pronounced under heavier workloads. The concurrent throughput gap grows from 1.5x at 5M to 2.4x at 100M at each engine's best-case memory and widens to 12x at 100M (16 GB `shared_buffers`, 64 GB total RAM) and 6.5x at 1B (32 GB `shared_buffers`, 128 GB total RAM) on our test hardware. Organizations that expect their data or query volume to grow will see increasing returns from VectorChord over time.

[1] These labels refer to the number of vectors, not the volume of data. Because the 1B benchmark's dataset has smaller vectors than the 100M benchmark's, the total volume grows by only ~25% from 100M to 1B, not 10x.

Scope note: This report compares pgvector's HNSW index with VectorChord's IVF-RaBitQ index on a single hardware tier. Read further for full methodology, measurement caveats, and follow-up items.

When to choose VectorChord: For datasets exceeding available memory, concurrent workloads, fast index builds, or applications requiring predictable latency under load, VectorChord delivers decisive advantages across the test scales (5M to 1B vectors): 1.5x–2.4x higher throughput at equivalent memory, 5x–12x faster builds, and P99 latency 50%–75% tighter under concurrent load.

When to choose pgvector HNSW: This report documents where VectorChord's architecture wins, but pgvector HNSW remains a sensible default for several workloads. For small datasets (≤5M vectors) and light concurrency, pgvector HNSW is well established in the PostgreSQL ecosystem and delivers adequate performance. For applications targeting very high recall (>0.98) where latency is secondary, HNSW's ability to trade `ef_search` for recall can be preferable to IVF's centroid-quality ceiling. See §6. Conclusion for the full discussion of when each engine is the right fit.

2. System specifications

Component	Specification
Instance	AWS EC2 i7i.metal-48xl
CPU	Intel Xeon Platinum 8559C, 96 cores / 192 threads, 320 MiB L3 cache
Memory	1,511 GB DDR5
Storage	4x 3.41 TiB NVMe local instance SSDs, LVM striped, 13.64 TiB XFS (/data)
OS	RHEL 9.6—Linux 5.14.0-570.72.1
PostgreSQL	17.9
pgvector	0.8.2
VectorChord	1.0.0

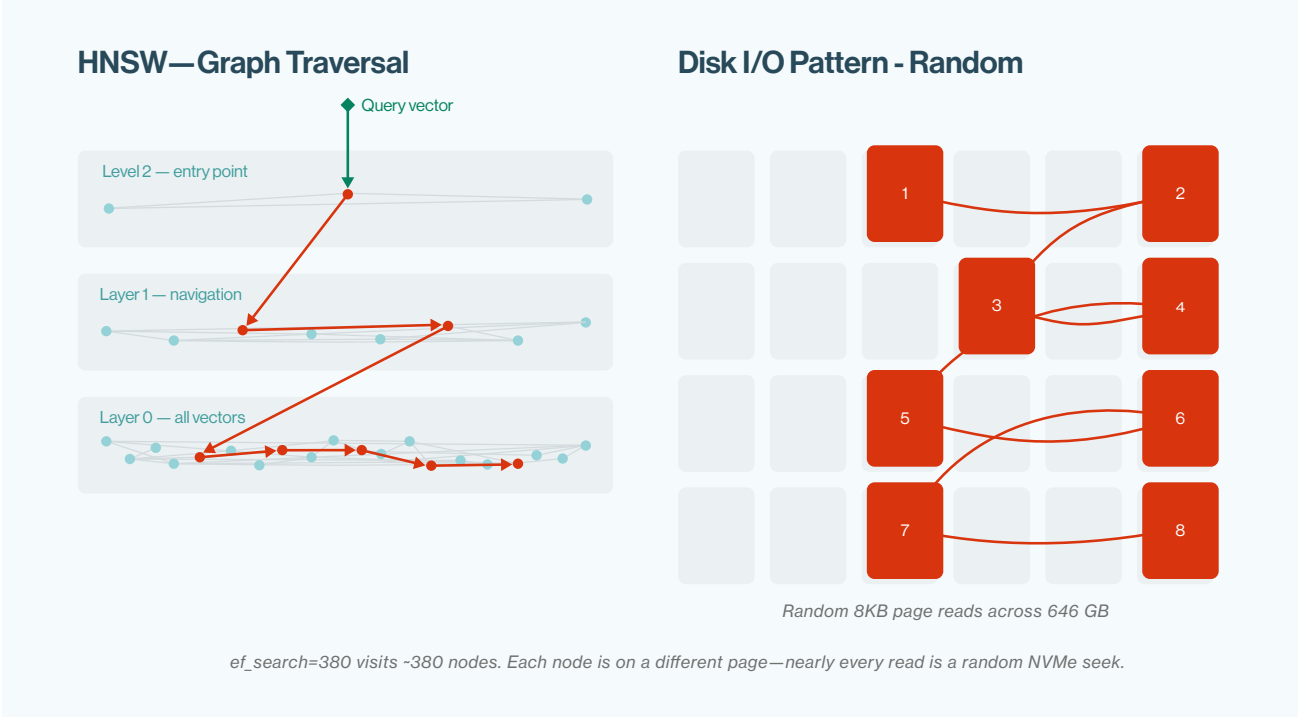
Datasets

Dataset	Vectors	Dimensions	Metric	Source
LAION 5M	5,000,000	768	Inner product	LAION-5B subset
LAION 100M	100,000,000	768	Inner product	LAION-5B subset
Deep1B	1,000,000,000	96	L2	Yandex deep 1B

3. How the indexes work

Understanding the I/O behavior of each index type is essential to interpreting the benchmark results. The fundamental difference is **random vs. sequential disk access**, which becomes the dominant performance factor when the index exceeds available memory.

pgvector—HNSW (hierarchical navigable small world)



ef_search=380 visits ~380 nodes. Each node is on a different page—nearly every read is a random NVMe seek.

HNSW organizes vectors into a multilayer graph. The top layers are sparse (few nodes, long-range connections), while lower layers are progressively denser. Layer 0 contains all vectors.

Query traversal:

1. Enter at the top layer's entry point.
2. Greedily traverse the graph—at each node, read its neighbor list (M neighbors), compute distances to each neighbor's vector, and move to the closest.
3. Descend to the next layer and repeat with a wider search.
4. At layer 0, expand the search to `ef_search` candidate nodes.

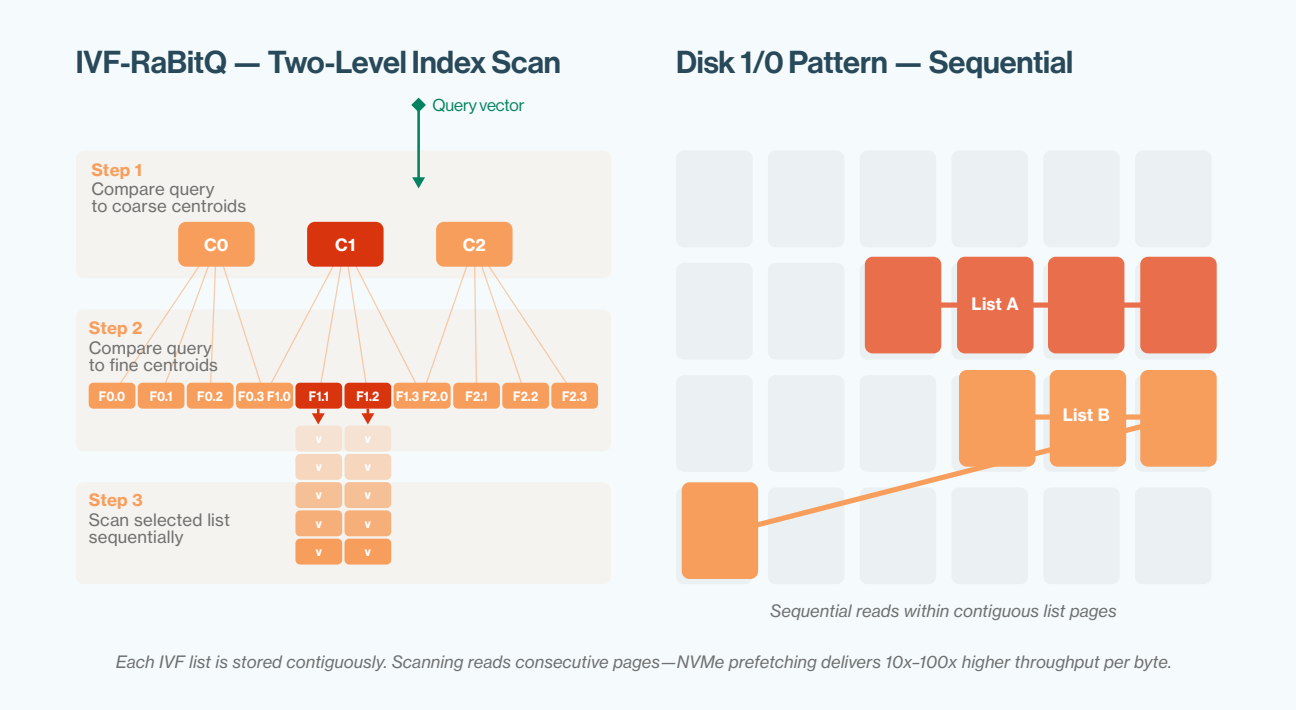
For `ef_search=380`, the query visits approximately 380+ nodes. Each node visit requires reading the node's vector and neighbor list from an 8KB PostgreSQL page. Because graph neighbors are determined by vector similarity rather than physical proximity, **consecutive nodes in the traversal are almost never on the same page**. This produces a random I/O pattern across the entire index.

I/O cost per query (1B dataset, ef = 380, 32 GB shared_buffers):

- ~380 node reads, each potentially a random page from 646 GB
- ~5% buffer cache hit rate (32 GB/646 GB)
- ~360 disk reads x ~2.5 ms NVMe random read latency = ~900 ms per query

This matches the observed P50 of 683 ms in our benchmarks.

VectorChord—IVF-RaBitQ (inverted file with randomized binary quantization)



Each IVF list is stored contiguously. Scanning reads consecutive pages—NVMe prefetching delivers 10x–100x higher throughput per byte.

VectorChord uses a two-level inverted file index built with **hierarchical k-means clustering**. Rather than clustering all vectors into a flat list of partitions (which would require computing distances to hundreds of thousands of centroids at query time), the dataset is organized into a two-level tree:

1. **Coarse level:** The dataset is first partitioned into a small number of coarse clusters also called centroids (e.g., 800 for 1B vectors), using k-means. Each coarse centroid represents a broad region of the vector space.
2. **Fine level:** Within each coarse partition, vectors are further clustered into many fine-grained lists (e.g., 640,000 total for 1B vectors). Each fine list contains vectors that are close neighbors, stored contiguously on disk.

This hierarchy is what makes routing efficient at scale. A single-level IVF with 640,000 lists would require computing distances to all 640,000 centroids for every query—prohibitively expensive. The two-level structure reduces this to ~800 coarse distance computations followed by fine-level computations only within the selected coarse partitions. Each vector is also compressed using RaBitQ quantization (1 bit per dimension) for fast approximate distance computation during list scanning.

Query traversal:

1. **Coarse routing:** Compute distances from the query to all coarse-level centroids (e.g., 800) and select the nearest ones (e.g., 40, the first value in `nprobe`). This is fast—only hundreds of distance computations against centroids fit entirely in memory.
2. **Fine routing:** Within each selected coarse partition, compute distances to its fine-level centroids and select the nearest fine lists (e.g., 250 total, the second value in `nprobe`). This narrows the search from 640,000 lists down to 250.
3. **List scanning:** For each selected fine list, sequentially read the RaBitQ-quantized vectors (1 bit per dimension), compute approximate distances using SIMD, and collect the top-K candidates.
4. **Reranking:** Refine the candidate set using full-precision vectors stored alongside the quantized codes. The `epsilon` parameter controls how aggressively candidates are pruned before this step (lower = faster but riskier, higher = more thorough).

The critical difference: **List scans are sequential**. All vectors in a list are stored contiguously on disk. When VectorChord scans a list, it reads consecutive pages—the OS prefetcher and NVMe controller can issue large sequential reads, which are 10x–100x faster than random reads per byte transferred.

Example: I/O cost per query (1B dataset, nprobe = 40,250, 32 GB shared_buffers)

- **Steps 1–2 (routing):** Distance computation against centroids—nearly free, centroids fit entirely in memory.
- **Step 3 (list scanning):** ~250 fine lists read from disk. Each list is stored as contiguous pages, so the OS prefetcher and NVMe controller serve them as large sequential reads at **3-5 GB/s**.
- **For comparison, pgvector HNSW:** ~380 random 8 KB page reads scattered across 646 GB of index, served at effectively **~500 MB/s**.
- **Result:** Both engines read a similar volume of data per query, but VectorChord reads it **6x–10x faster**—sequential I/O is fundamentally more efficient on any storage device.

This architectural difference explains why VectorChord maintains usable performance when the index exceeds available memory, while pgvector's throughput collapses.

Dimension limits

pgvector's HNSW stores the full vector, neighbor list, and metadata in a single index tuple, which must fit within a PostgreSQL 8KB page. At M=16, the vector alone consumes `4 × dim` bytes, leaving room for roughly 2,000 dimensions before hitting the page size ceiling. This is a physical constraint of PostgreSQL's storage layer, not a tunable parameter—raising the compile-time `HNSW_MAX_DIM` flag does not help if the resulting tuples exceed the page size. Supporting higher dimensions would require recompiling PostgreSQL with a larger block size (16KB+), a nonstandard configuration that affects the entire database.

This limits pgvector's ability to index output from several current embedding models that produce vectors above 2,000 dimensions:

Model	Dimensions
NV-Embed-v2 (NVIDIA)	4,096
Gemini Embedding (Google)	3,072
text-embedding-3-large (OpenAI)	3,072
Jina Embeddings v3	2,048
Cohere Embed v4.0	2,048

VectorChord does not have this constraint. RaBitQ quantized codes use 1 bit per dimension—at 4,096 dims, the quantized representation is only 512 bytes per vector, well within page limits. Full-precision vectors for reranking are stored separately. VectorChord supports up to 65,535 dimensions.

Note: We did not benchmark dimensions above 768 in this report. The dimension limits described above are architectural, not empirically validated at the extremes.

4. Methodology

Scope and limitations

This benchmark compares pgvector's HNSW index against VectorChord's IVF-RaBitQ index on a single hardware tier. All measurements were taken on that machine. When we report "memory-constrained" results, we used Linux huge pages to shrink the OS page cache on the same machine; CPU count, memory bandwidth, and NVMe bandwidth were not scaled down. Performance on smaller cloud instances where CPU and storage also scale down has not been measured and should not be inferred from the memory-sensitivity curves in this report.

Not in scope: pgvector's `ivfflat` index, quantization + rerank pipelines on pgvector (`halfvec, bit`), validated benchmarks on smaller cloud instances where CPU, RAM, and storage scale together, and write-path performance.

PostgreSQL configuration

Non-default settings applied for all tests:

Parameter	Value	Rationale
`shared_buffers`	varied per test	Primary variable under test
`maintenance_work_mem`	varied per build	Sized to avoid swap (see note below)
`max_parallel_maintenance_workers`	32	Parallel index builds (see note below)
`max_parallel_workers`	64	
`max_worker_processes`	128	
`effective_io_concurrency`	200	NVMe-appropriate
`random_page_cost`	1.1	NVMe-appropriate
`max_wal_size`	500 GB	Avoid checkpoints during builds
`autovacuum`	off	Eliminate background interference
`shared_preload_libraries`	vchord	Required for VectorChord

Note on maintenance_work_mem: pgvector's HNSW build requires the entire graph in memory. `maintenance_work_mem` was sized per build to match the estimated graph memory and avoid page swapping:

Build	maintenance_work_mem
pgvector 5M M16-64	32 GB
pgvector 5M M16-128	32 GB
pgvector 100M M16-64	400 GB
pgvector 100M M16-128	400 GB
pgvector 1B M16-64	1024 GB
pgvector 1B M16-128	1024 GB
VectorChord (all)	64 GB (default)

VectorChord's IVF-RaBitQ build does not require loading the full index into memory, so the PostgreSQL default of 64 MB was sufficient for all dataset sizes.

Note on parallel workers: While the system has 192 logical cores, `max_parallel_maintenance_workers` was set to 32. Both pgvector and VectorChord show diminishing returns beyond this point—the index build phases that benefit from parallelism (tuple loading, distance computations) become bottlenecked by synchronization overhead, memory bandwidth, and single-threaded phases (graph construction for HNSW, centroid computation for IVF). In our testing, scaling from 32 to 64+ workers yielded less than 5% improvement in build time while consuming significantly more memory.

Index configurations

pgvector (HNSW):

- M = 16 with ef_construction = 64 (faster build, lower recall)
- M = 16 with ef_construction = 128 (slower build, higher recall)

VectorChord (IVF-RaBitQ):

- Two list configurations per dataset (smaller/larger), using hierarchical k-means, residual quantization, and sampling factor 256

Dataset	pgvector configs	VectorChord configs
5M	M16/ef64, M16/ef128	[50, 8k], [190, 35k]
100M	M16/ef64, M16/ef128	[400, 160k], [570, 320k]
1B	M16/ef64, M16/ef128	[400, 160k], [800, 640k]

Test protocol

For each index configuration:

1. **Build phase:** Index built from scratch, recording wall-clock build time and final on-disk index size.
2. **Query phase:** 1,000 queries (top-10) per benchmark point, tested with both 1 client (sequential) and 32 concurrent clients.
3. **Memory tier simulation:** To evaluate sensitivity to available RAM, we used Linux huge pages to constrain the OS page cache on our 1.5 TB machine. Each tier locks away excess RAM, leaving only the target amount for `shared_buffers` + OS file system cache. **CPU count, memory bandwidth, and NVMe bandwidth were not scaled down**—results describe performance as a function of RAM on our test hardware, not performance on smaller cloud instances where CPU and storage would also be reduced.
4. **Search parameters:** 5–6 values per engine, targeting recall from ~0.90 to ~0.97, with one point calibrated to hit ~0.95 recall. For pgvector, the search parameter is `ef_search` (number of candidates explored during graph traversal). For VectorChord, the parameters are `nprob` (number of IVF lists probed at each level, specified as `coarse`,`fine`) and `epsilon` (reranking factor, fixed at 1.9 for all tests).

Memory tiers tested (available RAM = `shared_buffers` + OS page cache):

Dataset	Available RAM
5M (19–21 GB index)	8, 16, 32, 64, 128, 256 GB
100M (367–405 GB index)	64, 128, 256, 512, 1024, 1511 GB
1B (469–646 GB index)	128, 256, 512, 1024, 1200, 1511 GB

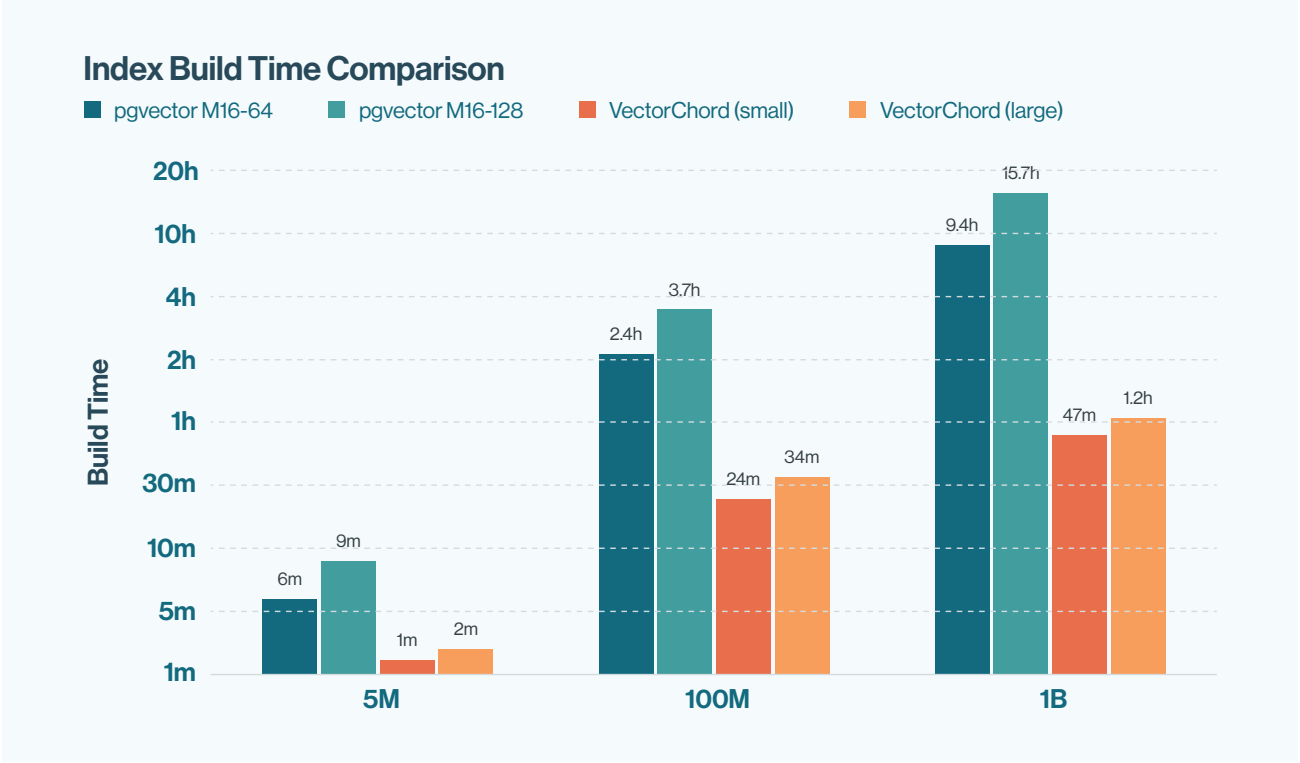
JIT compilation was disabled. Autovacuum was disabled. `enable_seqscan = off` for pgvector prevented the query planner from incorrectly choosing sequential scans at high ef_search values.

Benchmark suite

All benchmarks in this report were run using the **Vector Search Benchmark Tool (VSBT)**, an open source benchmarking suite developed by EnterpriseDB (EDB) and publicly available at github.com/EnterpriseDB/vsbt. VSBT supports pgvector, VectorChord, and GPU-accelerated builds (pgpu) across datasets from 1M to 1B vectors. It handles dataset loading, index creation, query benchmarking with configurable concurrency, server memory simulation via huge pages, and automated report generation. All configuration files used in this report are included in the repository—results can be reproduced on equivalent hardware by running the provided configs against the same datasets.

5. Results

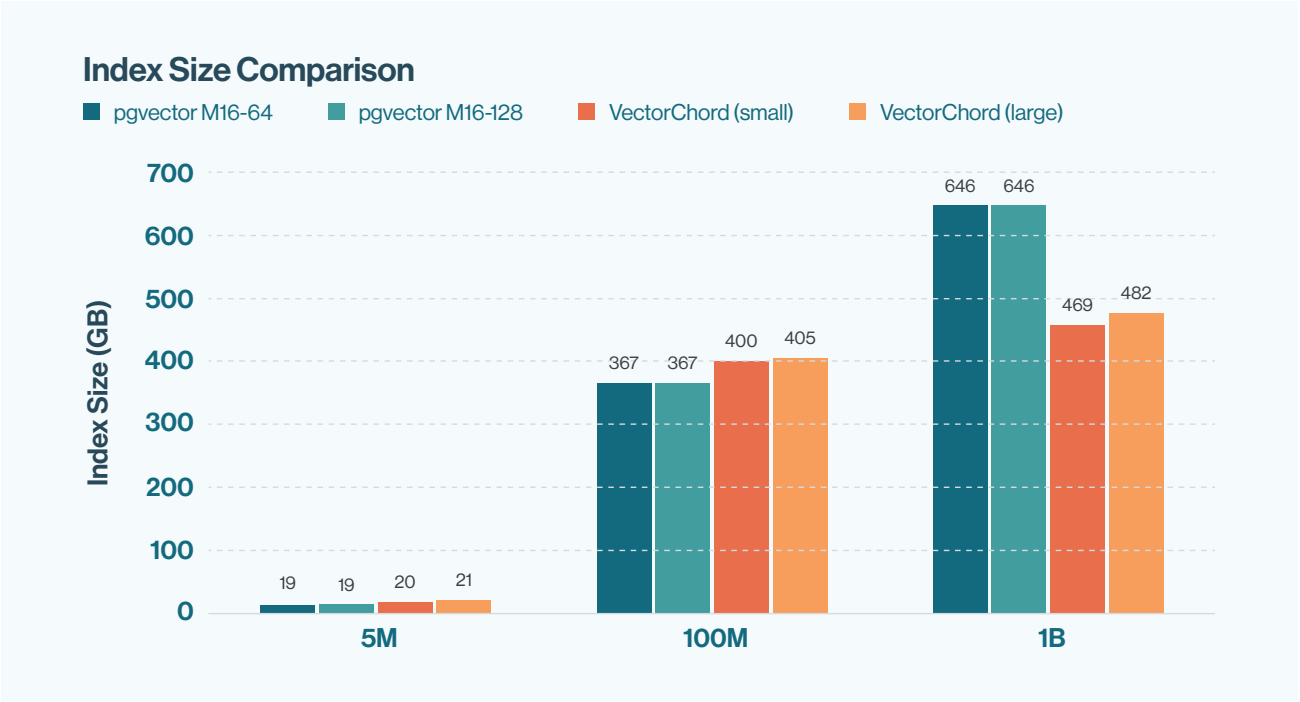
Build time



Dataset	pgvector M16-64	pgvector M16-128	VC (small lists)	VC (large lists)	Speedup (best vs. best)
5M	352s	558s	65s	102s	**5.4x**
100M	8,580s (2.4h)	13,391s (3.7h)	1,462s (24m)	2,140s (36m)	**5.9x**
1B	33,957s (9.4h)	56,466s (15.7h)	2,751s (46m)	4,232s (1.2h)	**12.3x**

VectorChord consistently builds 5x–13x faster than pgvector HNSW. The gap widens at a larger scale because HNSW construction requires searching the existing graph for each vector insertion, making build time grow faster than linearly with dataset size. IVF-RaBitQ's k-means sampling and quantization scale more predictably.

Index size

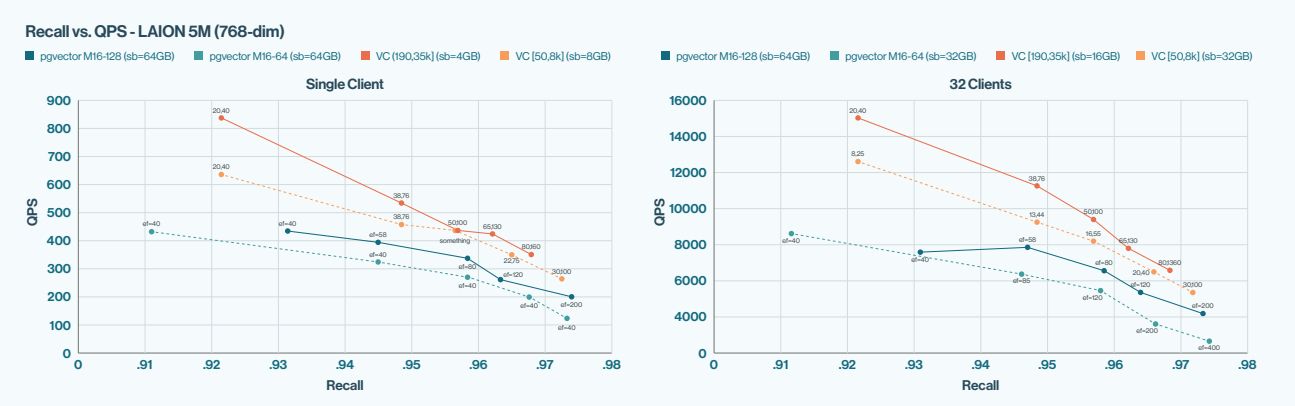


Dataset	pgvector (M16-64/M16-128)	VectorChord (small/large)	Ratio
5M	19/19 GB	20/21 GB	VC ~10% larger
100M	367/367 GB	400/405 GB	VC ~10% larger
1B	646/646 GB	469/482 GB	VC 25% smaller

Within each engine, index size is virtually identical regardless of configuration—`ef_construction` does not affect pgvector's on-disk size (only build speed and graph quality), and VectorChord's different list counts produce nearly the same storage footprint.

At 5M and 100M (768-dim vectors), index sizes are comparable across engines—raw vector data dominates both formats. At 1B (96-dim), the dynamic shifts: pgvector's HNSW graph edges (M = 16, ~32 neighbors per node across layers) dominate storage, while VectorChord's RaBitQ quantization compresses the low-dimensional vectors aggressively, resulting in a significantly smaller index.

Query performance: 5M dataset



At 5M vectors (19-21 GB indexes), the working set fits in memory across most RAM configurations tested. Both engines perform well, with VectorChord holding a consistent advantage.

Best results at ~0.95 recall, single client:

Engine	Config	Recall	QPS	P50	P99	shared_buffers
pgvector	M16-128, ef = 58	0.946	395	2.3 ms	6.1 ms	64 GB
pgvector	M16-64, ef = 85	0.945	323	2.7 ms	7.9 ms	64 GB
VectorChord	**[190,35k], 38,76**	**0.949**	**526**	**1.7 ms**	**4.2 ms**	**8 GB**
VectorChord	[50,8k], 13,44	0.949	474	1.9 ms	4.6 ms	8 GB

Best results at ~0.95 recall, 32 clients:

Engine	Config	Recall	QPS	P50	P99	shared_buffers
pgvector	M16-128, ef = 58	0.946	7,800	2.9 ms	7.5 ms	64 GB
VectorChord	**[190,35k], 38,76**	**0.949**	**11,361**	**1.8 ms**	**2.5 ms**	**16 GB**

VectorChord [190,35k] is **33% faster single-client and 45% faster under concurrency**. It peaks at just 8 GB `shared_buffers` while pgvector HNSW needs 64 GB. Under 32-client load, VectorChord's P99 is 3x tighter (2.5 ms vs. 7.5 ms).

Query performance: 100M dataset

At 100M vectors (367–405 GB indexes), the index far exceeds any practical shared_buffers size. Available memory becomes the critical factor.

Best results at ~0.95 recall, single client:

Engine	Config	Recall	QPS	P50	P99	shared_buffers
pgvector	M16-128, ef = 145	0.952	95	10.6 ms	20.1 ms	128 GB
VectorChord	**[570,320k], 17,140**	**0.951**	**187**	**5.3 ms**	**7.3 ms**	**32 GB**
VectorChord	[400,160k], 50,100	0.950	149	6.6 ms	9.2 ms	32 GB

Best results at ~0.95 recall, 32 clients:

Engine	Config	Recall	QPS	P50	P99	shared_buffers
pgvector	M16-128, ef = 145	0.952	1,917	16.5 ms	33.0 ms	256 GB
VectorChord	**[570,320k], 17,140**	**0.951**	**4,644**	**6.8 ms**	**8.9 ms**	**16 GB**

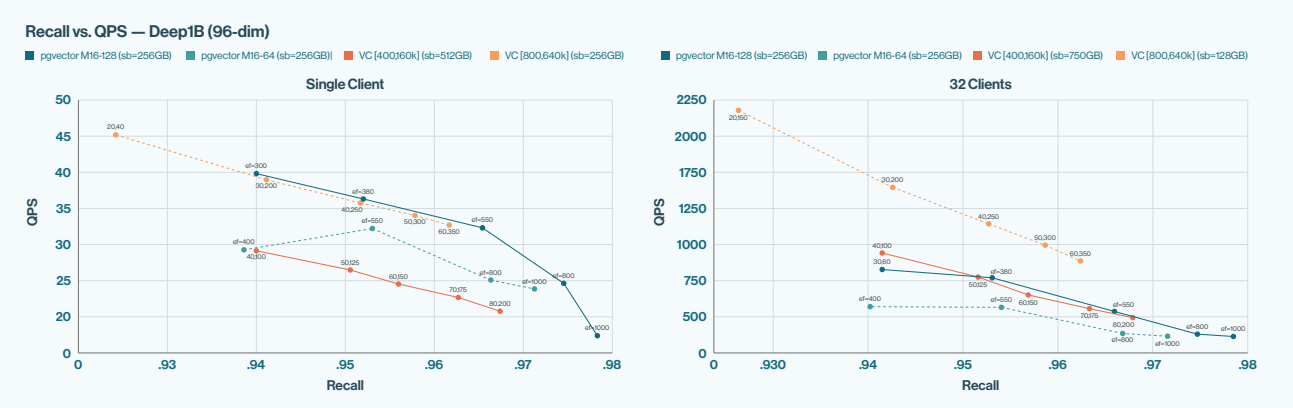
VectorChord is **2x faster single-client and 2.4x faster at 32 clients**, while needing a fraction of the memory. The P99 gap under concurrency is striking: **8.9ms vs. 33.0ms**.

With 16 GB `shared_buffers` and 64 GB total available RAM (index size 367–405 GB):

Engine	Config	QPS (1c)	QPS (32c)
pgvector	M16-128, ef = 145	13	377
VectorChord	**[570,320k], 17,140**	**179**	**4,644**

At this `shared_buffers` setting, VectorChord delivers 13.4x the single-client QPS (179 vs. 13) and 12.3x the 32-client QPS (4,644 vs. 377) of pgvector HNSW. The 367 GB pgvector HNSW index far exceeds the 64 GB of total RAM on our test hardware, so its random-read access pattern incurs disk latency on most reads; VectorChord's sequential list scans remain page-cache friendly at the same configuration.

Query performance: 1B dataset (Deep1B)



At 1B vectors (469–646 GB indexes), memory constraints dominate performance for both engines.

Best results at ~0.95 recall, single client:

Engine	Config	Recall	QPS	P50	P99	shared_buffers
pgvector	M16-128, ef = 380	0.952	36	27.9 ms	45.5 ms	256 GB
pgvector	M16-64, ef = 550	0.953	32	31.2 ms	54.5 ms	256 GB
VectorChord	**[800,640k], 40,250**	**0.952**	**36**	**25.2 ms**	**59.0 ms**	**128 GB**
VectorChord	[400,160k], 50,125	0.950	27	34.9 ms	77.7 ms	128 GB

Best results at ~0.95 recall, 32 clients:

Engine	Config	Recall	QPS	P50	P99	shared_buffers
pgvector	M16-128, ef = 380	0.952	818	40.6 ms	66.9 ms	256 GB
VectorChord	**[800,640k], 40,250**	**0.952**	**1,267**	**24.9 ms**	**35.7 ms**	**128 GB**

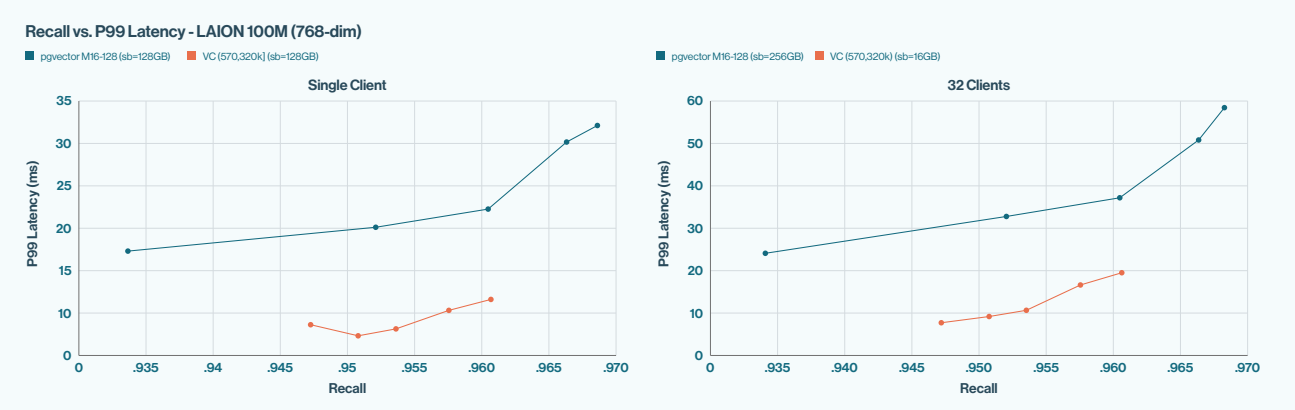
At 1B, single-client QPS is effectively identical (36 vs. 36), but VectorChord wins concurrent throughput by 55% (1,267 vs. 818 QPS) with half the P99 latency (35.7ms vs. 66.9ms). VectorChord achieves this at 128 GB `shared_buffers` while pgvector HNSW needs 256 GB.

With 32 GB `shared_buffers` – 128 GB total available RAM – on our test hardware (index size 469–646 GB):

Engine	Config	QPS (1c)	QPS (32c)	QPS (32c)
pgvector	M16-128, ef = 145	13	149	683 ms
VectorChord	[570,320k], 17,140	179	974	29 ms

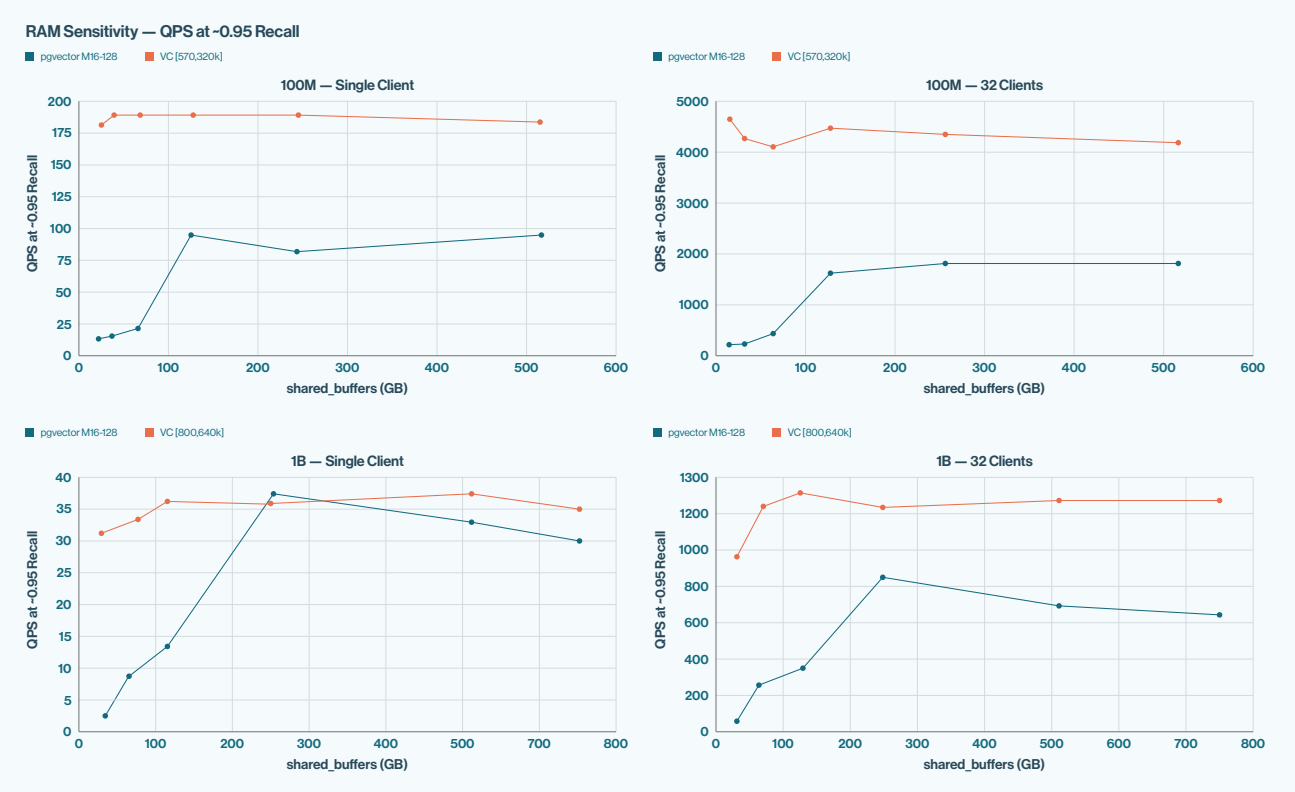
At this `shared_buffers` setting, VectorChord **delivers 20.7x the single-client QPS** (31 vs. 1.5) and **6.5x the 32-client QPS** (974 vs. 149) of pgvector HNSW. pgvector HNSW's single-client P50 at this configuration is 683 ms, and `pg_stat_activity` showed all 32 backends blocked on `BufferIo` wait events under 32-client load—CPUs idle while the random-read pattern against a 646 GB index saturated the storage path.

P99 latency



Across all datasets, VectorChord consistently delivers 50%–75% lower P99 latency than pgvector HNSW under concurrent load at equivalent recall. HNSW's graph traversal has inherently variable path length—some queries traverse short paths while others explore deep branches. Under concurrency, this variability compounds as clients contend for random I/O. IVF scans a fixed number of lists regardless of query difficulty, producing tighter latency distributions even under load.

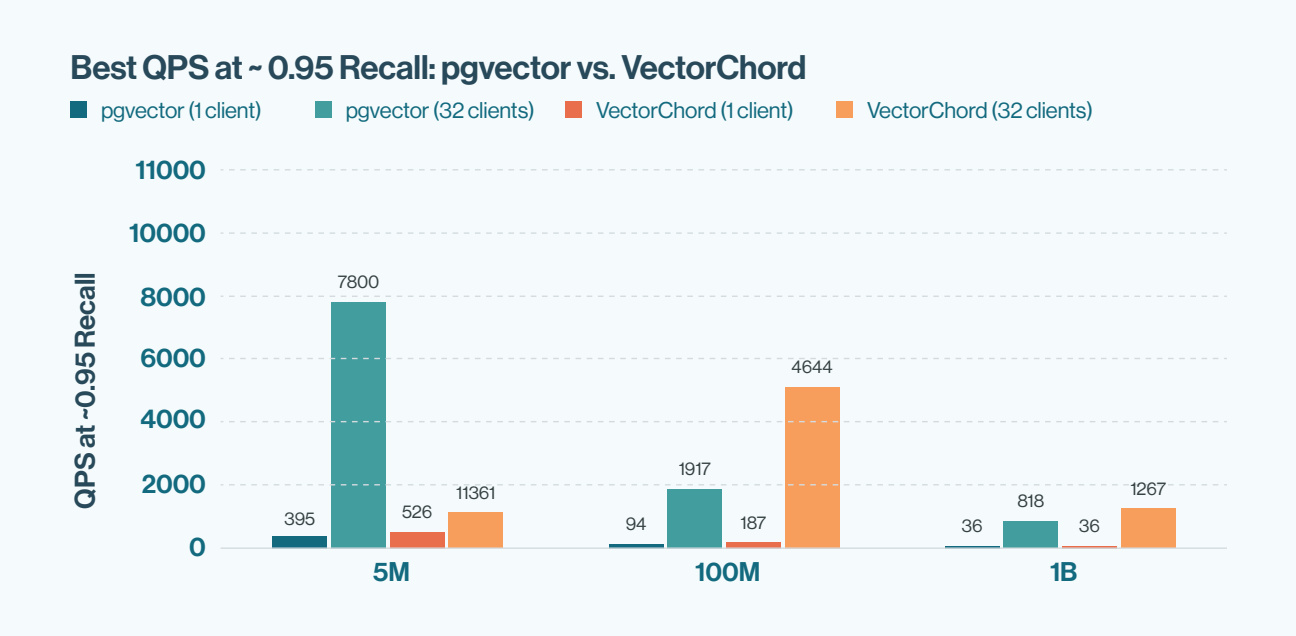
RAM sensitivity



VectorChord is nearly flat across available-RAM configurations. At 100M, single-client QPS varies only from 179 to 187 across the range from 16 GB to 512 GB `shared_buffers`. Under 32-client load, it sustains 4,100-4,600 QPS regardless of RAM configuration.

pgvector HNSW requires precise memory sizing. At 100M, single-client QPS ranges from 13 (16 GB `shared_buffers`) to 95 (128 GB)—a 7x spread. Under concurrency, the range is 377 to 1,917—a 5x spread. At 1B, pgvector HNSW peaks at 256 GB `shared_buffers`, then declines at 512–750 GB—a counterintuitive result in which more shared memory yields lower throughput, driven by the double-buffering effect described below.

Best QPS at ~0.95 recall across all RAM configurations tested



Why more shared_buffers can hurt: The double-buffering effect

When a PostgreSQL backend reads a page not found in `shared_buffers`, the kernel first loads it into the OS page cache, then copies it into `shared_buffers`—the same data now resides in memory twice. With small `shared_buffers` (e.g., 16-32 GB), nearly all RAM is available to the page cache, which can cache the full index in a single copy. With `shared_buffers` large enough for the entire index, every access is a direct `shared_buffers` hit with no kernel involvement—the fastest path. But at intermediate sizes, both caches hold partial, overlapping copies of the index. When total available memory is less than the index size, these duplicate pages reduce the total amount of unique index data that fits in RAM, increasing the disk I/O rate. The PostgreSQL buffer manager also adds per-access overhead—hash-table lookups, buffer pin/unpin, and clock-sweep victim selection—that the OS page cache avoids. Under HNSW's random access pattern, in which each query touches hundreds of pages scattered across the full index, this overhead compounds and produces measurably lower throughput than simply letting the page cache handle everything.

In practice, either size `shared_buffers` to hold the full index (eliminating page cache involvement), or keep it minimal and let the OS page cache manage the working set. The middle ground wastes memory on duplicate copies and adds buffer management overhead for every miss. This applies to both pgvector HNSW and VectorChord, though VectorChord's sequential I/O pattern makes it far less sensitive to the effect.

6. Conclusion

The benchmarks in this report reveal a consistent pattern: At small scale and light concurrency, pgvector HNSW is a capable starting point; as datasets grow and query concurrency increases, VectorChord's sequential I/O architecture pulls decisively ahead on throughput, latency stability, and memory efficiency. No single index type is the right answer for every workload—the right answer is a platform that supports both and lets organizations move between them as their requirements evolve.

EDB Postgres AI provides exactly that foundation: a production-grade, enterprise-ready PostgreSQL platform with native support for both extensions, built to serve agentic workloads from initial deployment through billion-vector scale. The guidance below is intended to help teams make the choice with confidence.

When pgvector HNSW may be sufficient

- **Small datasets that fit in memory:** At 5M vectors, pgvector delivers reasonable performance, though VectorChord is still 33%–45% faster in our tests. If the workload is light and the dataset is small, pgvectorHNSW's simpler configuration surface may be adequate.
- **Very high recall requirements (>0.98):** HNSW can push recall higher by increasing `ef_search`, while IVF recall has an upper bound determined by centroid quality. However, in our tests pgvector did not reach 0.98 recall even at `ef_search=1000`, and QPS dropped by more than 50% in the attempt. Achieving >0.99 recall with HNSW likely requires higher M values (denser graph), which would further increase index size, build time, and memory requirements.

When to use VectorChord

- **Datasets exceeding available memory:** VectorChord's sequential I/O pattern degrades gracefully, while pgvector HNSW's random I/O causes order-of-magnitude throughput collapse.
- **Concurrent workloads:** VectorChord scales nearly linearly with clients; pgvector HNSW's scaling is limited by I/O contention.
- **Build time matters:** 5x–12x faster index builds, critical for large datasets for which pgvector builds take hours.
- **Predictable latency:** 50%–75% tighter P99 under concurrent load, important for SLA-bound applications.
- **Memory-efficient deployments:** Lower `shared_buffers` requirements and smaller indexes at low dimensions (25% smaller at 1B, 96-dim) mean VectorChord can serve the same workload on less RAM.
- **Production workloads at any scale:** Even at 5M vectors, VectorChord outperforms pgvector on throughput, latency, and memory efficiency. The performance gap only widens as datasets grow.

7. Appendix A: Key considerations and caveats

A.1. Operational considerations

pgvector's HNSW index requires significant `maintenance_work_mem` during build (up to 1 TB for 1B vectors) but benefits from large `shared_buffers` during query serving. If `maintenance_work_mem` is undersized—i.e., the in-memory graph exceeds the allocation mid-build—pgvector flushes the graph to disk and switches to an on-disk insertion mode for the remaining vectors. This is a one-way transition that emits only a NOTICE (`hnsw graph no longer fits into maintenance_work_mem`), not an error. The resulting build time can increase by an order of magnitude or more, as every remaining vector insertion requires random reads and writes through the buffer manager instead of direct memory access. There is no graceful spill or recovery—the earlier the overflow, the worse the impact.

Since `shared_buffers` requires a PostgreSQL restart to change, this creates an additional operational burden in production: You must restart the database after building the index to reclaim memory and resize the buffer pool. This means downtime or a failover during the transition from build to query-serving mode.

VectorChord does not have this constraint. Its build-time memory usage is modest regardless of dataset size (64 MB default `maintenance_work_mem`), and query performance is largely insensitive to `shared_buffers` sizing. No restart is needed between build and query phases.

`maintenance_work_mem` can be changed at session level without a restart, but the combination of needing high `maintenance_work_mem` for build and high `shared_buffers` for queries means pgvector deployments typically cannot optimize for both simultaneously without a restart cycle.

A.2. Index rebuilds in production

Indexes need to be rebuilt periodically—after major data changes, schema migrations, extension upgrades, or to improve quality as the dataset evolves. The build time and resource differences between the two engines have significant operational consequences.

pgvector HNSW at 1B vectors: A rebuild takes 15.7 hours (M16-128) and requires up to 1 TB of `maintenance_work_mem`. During the build, this memory allocation leaves little headroom for concurrent query serving. After the build completes, PostgreSQL must be restarted to reconfigure `shared_buffers` for query-optimized settings. The total maintenance window is the build time plus restart plus index warm-up—realistically, a weekend operation requiring planned downtime or a failover to a replica.

VectorChord at 1B vectors: A rebuild takes 1.2 hours and doesn't require `maintenance_work_mem`. The build can run on the same instance that serves queries without starving them of memory, and no restart is needed afterward. This is the difference between a weekend maintenance window and a rolling update during business hours.

At 100M vectors, the gap is proportionally similar: 3.7 hours vs. 36 minutes. For organizations that need to rebuild indexes regularly—whether due to data freshness requirements, A/B testing different index configurations, or disaster recovery—this difference compounds into a major operational advantage.

A.3. Index update overhead

Write workloads were not benchmarked in this report. The following is a theoretical analysis; quantitative validation is planned.

The two architectures have fundamentally different per-write costs. Inserting a vector into HNSW requires searching the graph to find neighbors (an `ef_construction`-bounded traversal), then updating neighbor lists across multiple scattered pages—the same random I/O pattern that limits query performance, applied to every write. Deletions leave dead nodes in the graph, degrading search quality until a full `REINDEX`.

Inserting into IVF requires only a centroid distance computation (centroids fit in memory) and appending to a single list—no graph traversal, no scattered writes. Deletions affect only the local list's scan time, not global search quality.

For workloads with frequent updates—real-time recommendation systems, continuously ingested embeddings, or high-churn applications—this difference in write overhead could be as operationally significant as the query performance gap documented above.

A.4. Limitations and caveats

HNSW parameter space: We tested only M = 16 with ef_construction = 64 and 128. Higher values of M (e.g., 32 or 48) would produce a denser graph with better recall at lower ef_search values, at the cost of larger indexes, slower builds, and significantly higher memory requirements during both build and query serving. This could narrow the query performance gap, particularly at high recall targets. Similarly, `ef_construction` values above 128 could improve graph quality further.

VectorChord tuning: VectorChord has more tuning parameters than pgvector (two-level list counts, sampling factor, epsilon, residual quantization). While we tested two list configurations per dataset, there may be configurations we did not explore that yield better results. The same applies to pgvector—our parameter sweep is not exhaustive.

Recall target: We focused on the ~0.95 recall range. At very high recall targets (>0.99), pgvector's HNSW can push `ef_search` arbitrarily high with predictable recall gains. VectorChord's IVF recall has an upper bound determined by centroid quality—probing more lists has diminishing returns past a point. For applications requiring >0.99 recall, the comparison may favor pgvector.

Index size at 1B: VectorChord's 25% smaller index at 1B (Deep1B, 96-dim) is dimension dependent. The approximate on-disk size formulas are:

- **pgvector HNSW:** `N x (4 x dim + 8 x M + C)`, where `8 x M` is the graph edge storage per node (neighbor IDs across layers) and C is per-node overhead (~64 bytes including page headers and item pointers)
- **VectorChord IVF-RaBitQ:** `N x (4 x dim + dim/8 + C) + centroids`, where `dim/8` is the RaBitQ quantized representation (1 bit per dimension) and C is per-entry overhead

At high dimensions (768-dim), the `4 x dim` raw vector term dominates both formulas and sizes converge. At low dimensions (96-dim), pgvector's fixed `8 x M` graph edge cost (128 bytes for M=16) becomes significant relative to the vector itself (384 bytes), inflating the index. VectorChord's quantized term (`dim/8` = 12 bytes at 96-dim) remains negligible. The size advantage should not be generalized across all dimensionalities.

Build machine requirements: pgvector's HNSW build required up to 1 TB of `maintenance_work_mem` for the 1B dataset. This means the build must run on a machine with sufficient RAM, which may not be the same as the query-serving machine. In practice, organizations may build the index on a larger temporary instance and restore the database to a smaller query-serving instance. VectorChord's 64 MB build requirement eliminates this operational complexity but doesn't make the comparison unfair—it reflects a genuine architectural difference.

Server simulation methodology: We used Linux huge pages to constrain available RAM, simulating smaller servers on our 1.5 TB machine. This provides a controlled environment where file system cache is naturally limited rather than artificially dropped. However, the NVMe storage, CPU, and memory bandwidth remain that of a high-end server—a true 128 GB server would also have fewer CPU cores and potentially slower storage, which could change the relative performance.

Query distribution: We use each dataset's standard test query set (ground truth queries). Production query distributions may be skewed, with some vectors or regions queried more frequently. Skewed distributions would favor caching and could change the relative performance of the two engines.

Statistical significance: All benchmarks use 1,000 queries per test point. While sufficient for trend analysis, the confidence intervals are wider than larger sample sizes. Recall values may vary by +/-0.002 between runs.

Software versions: Results are specific to pgvector 0.8.2 and VectorChord 1.0.0 on PostgreSQL 17.9. Both extensions are under active development and future versions may significantly change the performance characteristics.

8. Appendix B: Raw data

B.1. Build metrics

Test Name	Suite	Dataset	Index Config	Build Time	Index Size
pgvector-laion-5m-m16-64	pgvector	LAION 5M	M=16, ef_c=64	352s	19 GB
pgvector-laion-5m-m16-128	pgvector	LAION 5M	M=16, ef_c=128	558s	19 GB
vc-laion-5m-50-8k	VectorChord	LAION 5M	[50, 8000]	65s	20 GB
vc-laion-5m-190-35k	VectorChord	LAION 5M	[190, 35000]	102s	21 GB
pgvector-laion-100m-m16-64	pgvector	LAION 100M	M=16, ef_c=64	8,580s	367 GB
pgvector-laion-100m-m16-128	pgvector	LAION 100M	M=16, ef_c=128	13,391s	367 GB
vc-laion-100m-400-160k	VectorChord	LAION 100M	[400, 160000]	1,462s	400 GB
vc-laion-100m-570-320k	VectorChord	LAION 100M	[570, 320000]	2,140s	405 GB
pgvector-deep1B-m16-64	pgvector	Deep1B	M=16, ef_c=64	33,957s	646 GB
pgvector-deep1B-m16-128	pgvector	Deep1B	M=16, ef_c=128	56,466s	646 GB
vc-deep1B-400-160k	VectorChord	Deep1B	[400, 160000]	2,751s	469 GB
vc-deep1B-800-640k	VectorChord	Deep1B	[800, 640000]	4,232s	482 GB

B.2. Query benchmarks: Server tier sweep at ~0.95 recall

The tables below show the data points underlying sections 5.3-5.5. For each dataset, the search parameter closest to ~0.95 recall is shown across all simulated server sizes. “Available RAM” is the total RAM available to PostgreSQL on our test hardware (`shared_buffers` + OS file system cache), constrained via Linux huge pages. CPU count and NVMe bandwidth were not scaled down.

5M dataset (LAION, 768-dim, inner product)

pgvector M16-128, ef_search = 58 (recall 0.946):

Available RAM	shared_buffers	Clients	QPS	P50 (ms)	P99 (ms)
8 GB	2 GB	1	145.48	6.74	12.79
16 GB	4 GB	1	49.36	19.87	38.60
32 GB	8 GB	1	212.30	4.56	9.67
64 GB	16 GB	1	176.72	5.53	11.90
128 GB	32 GB	1	391.96	2.27	6.18
256 GB	64 GB	1	395.26	2.25	6.07
8 GB	2 GB	32	2,379.63	13.05	25.70
16 GB	4 GB	32	1,280.82	23.38	57.50
32 GB	8 GB	32	6,594.97	4.13	8.54
64 GB	16 GB	32	7,154.83	2.83	6.68
128 GB	32 GB	32	7,262.27	2.95	7.79
256 GB	64 GB	32	7,800.22	2.92	7.50

VectorChord [190, 35k], nprob = 38,76, epsilon = 1.9 (recall 0.949):

Server	shared_buffers	Clients	QPS	P50 (ms)	P99 (ms)
8 GB	2 GB	1	302.07	3.23	5.01
16 GB	4 GB	1	525.97	1.73	4.18
32 GB	8 GB	1	525.17	1.73	4.19
64 GB	16 GB	1	514.71	1.77	4.22
128 GB	32 GB	1	510.09	1.78	4.24
256 GB	64 GB	1	503.57	1.80	4.26
8 GB	2 GB	32	5,618.89	5.40	8.50
16 GB	4 GB	32	9,869.33	1.86	2.78
32 GB	8 GB	32	10,986.40	1.80	2.94
64 GB	16 GB	32	11,360.56	1.77	2.51
128 GB	32 GB	32	10,953.55	1.80	2.55
256 GB	64 GB	32	11,073.84	1.80	2.65

100M dataset (LAION, 768-dim, inner product)

pgvector M16-128, ef_search = 145 (recall 0.952):

Server	shared_buffers	Clients	QPS	P50 (ms)	P99 (ms)
64 GB	16 GB	1	13.33	75.30	133.30
128 GB	32 GB	1	15.48	64.75	114.71
256 GB	64 GB	1	21.45	46.89	79.64
512 GB	128 GB	1	94.46	10.61	20.11
1,024 GB	256 GB	1	81.10	12.26	25.00
1,511 GB	512 GB	1	94.14	10.39	24.48
64 GB	16 GB	32	377.21	84.09	149.48
128 GB	32 GB	32	390.79	81.60	144.86
256 GB	64 GB	32	585.28	54.93	96.15
512 GB	128 GB	32	1,728.34	18.47	31.80
1,024 GB	256 GB	32	1,916.84	16.48	32.98
1,511 GB	512 GB	32	1,892.80	16.54	37.81

VectorChord [570, 320k], nprob = 17,140, epsilon = 1.9 (recall 0.951):

Server	shared_buffers	Clients	QPS	P50 (ms)	P99 (ms)
64 GB	16 GB	1	178.63	5.52	7.93
128 GB	32 GB	1	186.68	5.30	7.25
256 GB	64 GB	1	186.27	5.32	7.30
512 GB	128 GB	1	186.91	5.29	7.34
1,024 GB	256 GB	1	186.52	5.29	7.73
1,511 GB	512 GB	1	183.21	5.37	8.35
64 GB	16 GB	32	4,644.39	6.79	8.94
128 GB	32 GB	32	4,278.71	7.39	9.71
256 GB	64 GB	32	4,123.66	7.65	9.99
512 GB	128 GB	32	4,461.83	7.06	9.49
1,024 GB	256 GB	32	4,359.47	7.22	9.50
1,511 GB	512 GB	32	4,195.35	7.47	10.37

1B dataset (Deep1B, 96-dim, L2)

pgvector M16-128, ef_search = 380 (recall 0.952):

Server	shared_buffers	Clients	QPS	P50 (ms)	P99 (ms)
128 GB	32 GB	1	1.54	682.73	876.03
256 GB	64 GB	1	8.82	118.89	165.43
512 GB	128 GB	1	13.49	78.41	104.94
1,024 GB	256 GB	1	36.28	27.94	45.49
1,200 GB	512 GB	1	33.01	29.97	53.88
1,511 GB	750 GB	1	29.73	32.88	60.69
128 GB	32 GB	32	148.75	226.73	299.77
256 GB	64 GB	32	260.89	129.41	173.09
512 GB	128 GB	32	365.59	92.03	128.37
1,024 GB	256 GB	32	818.42	40.56	66.90
1,200 GB	512 GB	32	676.40	47.15	88.33
1,511 GB	750 GB	32	659.65	47.96	95.36

VectorChord [800, 640k], nprob = 40,250, epsilon = 1.9 (recall 0.952):

Server	shared_buffers	Clients	QPS	P50 (ms)	P99 (ms)
128 GB	32 GB	1	31.03	29.24	65.40
256 GB	64 GB	1	33.20	27.30	63.10
512 GB	128 GB	1	35.65	25.12	58.97
1,024 GB	256 GB	1	35.71	25.17	59.05
1,200 GB	512 GB	1	35.59	25.37	59.04
1,511 GB	750 GB	1	34.89	25.87	60.06
128 GB	32 GB	32	973.51	31.43	62.84
256 GB	64 GB	32	1,226.02	25.74	35.49
512 GB	128 GB	32	1,266.72	24.88	35.72
1,024 GB	256 GB	32	1,228.45	25.71	37.71
1,200 GB	512 GB	32	1,240.39	25.32	36.45
1,511 GB	750 GB	32	1,252.35	25.17	34.55

EDB Postgres AI is the first open, enterprise-grade sovereign data and AI platform, with a secure, compliant, and fully scalable environment, on premises and across clouds. Supported by a global partner network, EDB Postgres AI unifies transactional, analytical, and AI workloads, enabling organizations to operationalize their data and LLMs where, when, and how they need them. For more information, visit enterprisedb.com