ChatGPT

Vector DBs for Local Photo Organizer

The use case is storing ~10k–50k **face embeddings** (e.g. 512-dim GhostFaceNet vectors) in a local photo app. All three options (ChromaDB, LanceDB, Postgres+pgvector) support fast ANN search (HNSW) and can handle this scale. The key differences are **deployment simplicity** and **index/storage design**. ChromaDB and LanceDB are *embedded* (no separate server), whereas PostgreSQL requires a running database service. All use HNSW for fast search (sub-10ms queries on ~10k vectors), but their storage differs.

- **ChromaDB (Local mode)** – an embedded Python library. ChromaDB stores vectors and metadata locally (by default in a `~/.chroma` directory using SQLite) and runs entirely on the host machine [1]. Under the hood it uses **hnswlib (HNSW)** for the ANN index and SQLite for metadata [2] [3]. This yields very fast search: in one benchmark on 5,000 vectors Chroma's HNSW lookup stayed ≈3 ms/query (37× faster than brute-force) [4]. (See graph below.) ChromaDB's index and data live in memory and on disk, with vector data persisted to Parquet/SQLite files. It is lightweight to install (`pip install chromadb`) and requires SQLite ≥ 3.35 [3]. **Drawbacks:** the HNSW index resides in RAM and *never shrinks*: deleting vectors leaves the index size high (space is only reclaimed by rebuilding the collection) [5]. Memory grows linearly with dataset size (e.g. ~90 MB total for 5k×1536-dim vectors [6]), but for 50k×512-dim it's still modest. ChromaDB is great for prototyping and local use [1] [4], but offers limited scalability beyond moderate sizes.

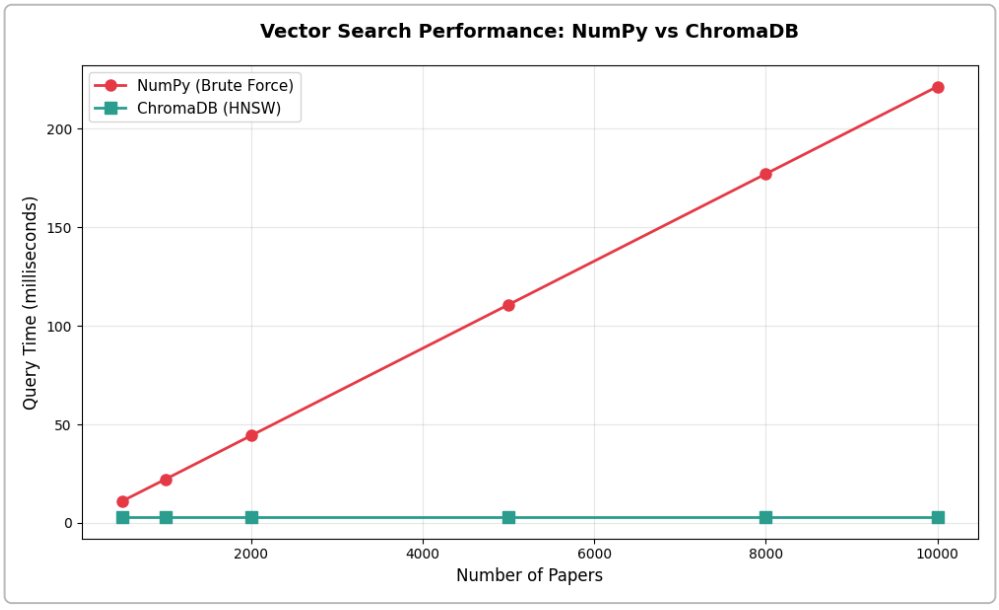

Figure: Search time vs. dataset size. ChromaDB's HNSW search (green) stays ~3 ms even as data grows to 10k vectors, whereas brute-force NumPy (red) grows linearly [4].

- **LanceDB** – an embedded, file-based Rust library. LanceDB saves all data in `.lance` files (a columnar "Lance" format) and requires **no server** [7] [8]. You simply do `lancedb.connect('path/to/file.lance')` in Python. By default LanceDB uses a *disk-backed*

IVF-PQ index for large-scale search [9], but it also offers an in-memory **HNSW** index for high-recall queries [10] [11]. Because it's disk-based, LanceDB's memory usage is very low; you can even query files on S3 without loading all vectors into RAM [8] [12]. Its Rust implementation is efficient and "well-suited for embedded or edge" use [13] [14]. In benchmarks LanceDB is extremely fast: e.g. <100 ms to search 1 billion 128-D vectors on a MacBook [15]. For 10–50k vectors its queries will be sub-millisecond even in Python, with throughput similar to Chroma's HNSW. **Pros:** no service to run, very fast on large data, flexible (mixes vector+scalar queries, versioning). **Cons:** newer project (smaller ecosystem), data stored in Lance format (not as ubiquitous as SQLite), requires building indexes (IVF/PQ) if you want disk efficiency.

- **PostgreSQL + pgvector** – a traditional SQL database with a vector extension. You define a `VECTOR` column and can add an **HNSW** or **IVFFlat** index on it [16]. HNSW in pgvector gives "lightning-fast searches" with excellent recall, though index builds are slower and use more memory [17]. Query syntax integrates with SQL (e.g. `ORDER BY embedding <-> queryvec LIMIT k`). This route is **service-based**: you must run a Postgres server process, configure pgvector, etc. This adds complexity for a pure local app. On the plus side, you get ACID transactions, joins, and familiar tooling. In practice, for 10–50k vectors a pgvector HNSW index will easily return nearest neighbors in a few milliseconds (especially if the index fits in RAM [18]). But you sacrifice the "single-file" simplicity. In short, pgvector works and scales to large data, but it's heavyweight for a small, single-user photo app.

Summary & Recommendation

All three options **support HNSW ANN search** and can handle 10k–50k vectors in memory. Their trade-offs are:

- **Speed (ANN search):** All use HNSW (or equivalent) so queries are extremely fast (~ms). ChromaDB's HNSW stays flat at ~3 ms for 5k vectors [4], LanceDB likewise delivers single-digit ms on much larger data [15], and pgvector/HNSW likewise is designed for sub-10ms search. (At this scale you likely won't notice major differences in speed.)

- **Simplicity (deployment):** ChromaDB and LanceDB are **embedded** (no service). Chroma runs as a local Python library (data in a local directory via SQLite) [1]. Lance runs as a local library reading/writing `.lance` files [7] [8]. By contrast, PostgreSQL requires running a database server, which is complex for a desktop app.

- **Storage format:** ChromaDB uses SQLite/Parquet under the hood (must have SQLite 3.35+ [3]). LanceDB uses its own columnar `.lance` format optimized for large vectors [9]. PostgreSQL stores vectors in its data files. All can persist to disk, but only Chroma/Lance do so in an "all-in-one" file manner.

- **Scalability & memory:** For 10–50k vectors, all handle easily. Chroma keeps the full HNSW graph in RAM (index grows but doesn't shrink [5]), so plan for a few hundred MB at most. LanceDB's default is disk-based IVFPQ (low RAM) but can build an in-memory HNSW if desired; it excels when scaling to millions (e.g. works directly on S3) [9] [15]. PostgreSQL's HNSW resides in memory (you can adjust `maintenance_work_mem` for build) [17].

- **Integration:** If you want a pure Python/local solution, Chroma or Lance is easier (just `pip install`). If your app already uses Postgres for other data, pgvector could be convenient, but otherwise it's overkill.

Conclusion: For a *local-first* face-organizer app, a lightweight embedded DB is preferable. ChromaDB and LanceDB both meet the criteria. ChromaDB offers a familiar Python/SQLite setup and is very easy to use for a few thousand vectors [1] [4]. LanceDB likewise runs locally in one file and can be faster on larger datasets (and supports features like versioning) [7] [15]. PostgreSQL + pgvector is powerful but requires running a full DB server, making it more complex for this use case. In practice, **ChromaDB or LanceDB** would be the "best" fit: both are *embedded, fast* vector stores. If absolute simplicity is key (pure Python/SQLite), ChromaDB is a solid choice [1] [19]; if you anticipate huge future scale or want a very file-centric approach, LanceDB is also compelling [7] [9].

Sources: Official docs and benchmarks for ChromaDB [1] [4], LanceDB [9] [15], and pgvector [20] [18]. These describe their architectures (HNSW indexing, storage) and performance. All content above is drawn from these sources.

---

[1] [4] [5] [6] Introduction to Vector Databases using ChromaDB – Dataquest
https://www.dataquest.io/blog/introduction-to-vector-databases-using-chromadb/

[2] [14] Vector databases (1): What makes each one different? • The Data Quarry
https://thedataquarry.com/blog/vector-db-1/

[3] How to Install and Use Chroma DB - DatabaseMart AI
https://www.databasemart.com/blog/how-to-install-and-use-chromadb?srsltid=AfmBOopSegPTQTm4JygV0BkSbFSzgRbDF1d44RD-Pwc06q9AeHWSP4ox

[7] Vector Databases: Lance vs Chroma | by Patrick Lenert - GenAI Fullstack Developer | Medium
https://medium.com/@patricklenert/vector-databases-lance-vs-chroma-cc8d124372e9

[8] A scalable, elastic database and search solution for 1B+ vectors built on LanceDB and Amazon S3 | AWS Architecture Blog
https://aws.amazon.com/blogs/architecture/a-scalable-elastic-database-and-search-solution-for-1b-vectors-built-on-lancedb-and-amazon-s3/

[9] [10] [11] Indexing Data - LanceDB
https://docs.lancedb.com/indexing

[12] [15] Benchmarking LanceDB. How to optimize recall vs latency for... | by Chang She | LanceDB | Medium
https://medium.com/etoai/benchmarking-lancedb-92b01032874a

[13] [19] Chroma vs LanceDB | Vector Database Comparison
https://zilliz.com/comparison/chroma-vs-lancedb

[16] [17] [20] GitHub - pgvector/pgvector: Open-source vector similarity search for Postgres
https://github.com/pgvector/pgvector

[18] Supercharging vector search performance and relevance with pgvector 0.8.0 on Amazon Aurora PostgreSQL | AWS Database Blog
https://aws.amazon.com/blogs/database/supercharging-vector-search-performance-and-relevance-with-pgvector-0-8-0-on-amazon-aurora-postgresql/