Performance Comparison of Chroma and pgvector: Vector Database Systems for Similarity Search

Λεωνίδας Φίλιππος Ανάγνου
AM: el20166, Team ID: 25
National Technical University of Athens
School of Electrical and Computer Engineering
el20166@mail.ntua.gr

***Abstract*—This report presents a detailed performance comparison between two widely-used open-source vector database solutions: Chroma and PostgreSQL with the pgvector extension. As vector databases have become essential infrastructure for machine learning applications, similarity search, and retrieval-augmented generation (RAG) systems, understanding their performance characteristics is critical for informed system design decisions. We evaluate both systems across data ingestion throughput, query latency, recall accuracy, storage efficiency, and resource utilization. Our experimental setup covers datasets ranging from 10,000 to 2,000,000 vectors with dimensionalities from 32 to 1,536. We test multiple index configurations including IVFFlat and HNSW with various parameter settings, and measure performance with and without metadata filters. To ensure fair comparison, both databases run in Docker containers with identical resource constraints (4GB memory, 6 CPUs). Our results show that Chroma achieves higher ingestion throughput (up to 7,000 vectors/sec vs 3,800 for pgvector) but uses more memory. For queries, pgvector demonstrates lower latency (sub-3 ms mean latency across all tested configurations) while Chroma shows higher recall for smaller datasets (e.g., 0.92 vs 0.87 recall@10 on 10k vectors). The complete benchmarking infrastructure is open-sourced for reproducibility.**

***Index Terms*—Vector databases, similarity search, Chroma, pgvector, performance benchmarking, approximate nearest neighbor, HNSW, IVFFlat, embedding vectors**

I. Introduction

VECTOR databases have emerged as critical infrastructure for modern artificial intelligence applications [9]. With the proliferation of large language models (LLMs) and embedding-based systems, the ability to efficiently store, index, and query high-dimensional vector data has become paramount. These systems enable similarity search operations that power recommendation engines, semantic search, retrieval-augmented generation (RAG), image recognition, and numerous other AI-driven applications.

Traditional relational databases are not optimized for the mathematical operations required for vector similarity computations. As a result, specialized vector database systems have been developed to address these requirements. Two notable approaches have emerged: purpose-built vector databases like Chroma, and extensions to existing database systems like pgvector for PostgreSQL.

A. Motivation and Objectives

The selection of an appropriate vector database system depends on multiple factors including performance requirements, scalability needs, operational complexity, and integration with existing infrastructure. This project aims to provide empirical data to support such decisions by conducting a rigorous comparison between Chroma and pgvector, following methodologies similar to those used in industry benchmarks [13], [14].

Our primary objectives are:

1) Deploy and configure both vector database systems in a reproducible environment with identical resource constraints
2) Generate synthetic datasets with varying sizes (10K to 2M vectors) and dimensionalities (32 to 1536)
3) Measure ingestion performance including throughput, index creation time, and resource consumption
4) Evaluate query performance across different top-K values with and without metadata filters
5) Measure recall accuracy against ground truth computed via brute-force search
6) Compare index types (IVFFlat vs HNSW) and their parameter configurations

B. Report Organization

The remainder of this paper is organized as follows: Section II provides background on vector databases and the systems under evaluation. Section III details our experimental methodology and fairness considerations. Section IV presents ingestion benchmark methodology and results. Section V analyzes query performance and recall. Section VI discusses limitations and future work. Section VII concludes with recommendations. Appendix A provides complete setup instructions.

II. Background and Related Work

A. Vector Similarity Search

Vector similarity search is the task of finding vectors in a database most similar to a query vector. For a query vector q and database $V = \{v_1, v_2, ..., v_n\}$, the goal is to find:

$$\text{NN}(q) = \arg\min_{v \in V} d(q, v) \tag{1}$$

where $d(\cdot, \cdot)$ is a distance function. Common distance metrics include Euclidean (L2) distance, cosine similarity, and inner product [11]. Both Chroma and pgvector support all three metrics.

For all benchmarks in this study, we use Euclidean (L2) distance as it is the most commonly used metric and provides consistent baseline comparison:

$$d_{L2}(q, v) = \sqrt{\sum_{i=1}^{D}(q_i - v_i)^2} \tag{2}$$

Exact nearest neighbor search becomes computationally prohibitive for high-dimensional data. Approximate Nearest Neighbor (ANN) algorithms trade accuracy for speed [10].

B. Index Structures

Various index structures have been developed for efficient ANN search, each with different trade-offs between build time, query speed, memory usage, and recall [12].

1) HNSW (Hierarchical Navigable Small World): HNSW [4] builds a multi-layer graph where each layer contains a subset of vectors. Key parameters:

- **M**: Maximum connections per node (affects memory and accuracy)
- **ef_construction**: Search depth during index building (affects build time and quality)

2) IVFFlat (Inverted File with Flat Quantization): IVFFlat [5] partitions vectors into clusters using k-means, then searches only relevant clusters. Key parameter:

- **lists**: Number of clusters (more lists = faster but less accurate)

C. Chroma

Chroma [1] is an open-source embedding database designed for AI applications, with growing adoption in the LLM ecosystem [18]:

- **Architecture**: Purpose-built with client-server model
- **Index**: Chroma uses HNSW as its index type. While Chroma does not expose index type selection, it allows tuning of key HNSW parameters (M, ef_construction) via collection metadata, which we leverage in our experiments.
- **API**: Python-first with collection-based abstractions
- **Metadata**: Native support for document metadata and filtering

D. pgvector

pgvector [2] extends PostgreSQL with vector capabilities. Detailed setup guides [15] and performance tuning recommendations [16] are available:

- **Architecture**: PostgreSQL extension leveraging existing RDBMS
- **Index Types**: Supports both IVFFlat and HNSW
- **SQL Integration**: Full SQL support with vector operators
- **Atomicity, Consistency, Isolation, Durability (ACID)**: Inherits PostgreSQL's transactional guarantees

III. Experimental Methodology

A. Hardware Configuration

All experiments were conducted on the following hardware:

TABLE I: Hardware Specifications

Component	Specification
CPU	Apple M4 Pro (14-core)
RAM	48 GB unified memory
Storage	1 TB NVMe SSD
Operating System	macOS Tahoe 26.2.0

B. Software Stack

The benchmarking infrastructure is implemented as a modular repository combining Python for core benchmarking logic with a Node.js/React web interface for interactive experimentation.

TABLE II: Software Components

Component	Technology	Version
Benchmark Scripts	Python	3.11+
Data Generation	NumPy	1.24+
Database Clients	chromadb, psycopg2	latest
Plotting	Matplotlib, Seaborn	latest
API Server	Node.js + Express	20.x
Web Frontend	React + Vite	18.x
Containerization	Docker Compose	v2

1) Repository Structure: The project is organized into distinct modules:

- **vec3/**: Core Python modules for data generation, ingestion, querying, and metrics computation
- **benchmarks/**: Benchmark runner scripts organized by type (ingestion, queries) and plotting utilities
- **scripts/**: Shell scripts for database management, benchmark orchestration, and automation
- **api/**: Node.js Express server providing REST endpoints for job management and results retrieval
- **web/**: React frontend for interactive benchmark configuration and visualization
- **results/**: Output directory for raw JSON results and generated
- **docs/**: Markdown documentation files

2) Key Dependencies: The Python environment relies on:

- `chromadb`: Official Chroma Python client for HTTP API access
- `psycopg2`: PostgreSQL adapter for pgvector connections
- `numpy`: Efficient vector operations and data generation
- `matplotlib/seaborn`: Result visualization and plot generation

The web interface uses:

- `Express`: Lightweight API server for benchmark job orchestration
- `React`: Interactive UI for parameter configuration
- `Vite`: Fast development server and build tooling

3) Version Control: git for version control and github projects for task management.

C. Docker Configuration and Resource Constraints

Both databases run as Docker containers with identical resource limits to ensure fair comparison:

Listing 1: Docker Compose Configuration

```yaml
services:
  pgvector:
    image: ankane/pgvector:latest
    container_name: pgvector_bench
    deploy:
      resources:
        limits:
          cpus: "6"
          memory: 4g
        reservations:
          cpus: "2"
          memory: 2g
    volumes:
      - pgvector_data:/var/lib/postgresql/data
    environment:
      POSTGRES_DB: vecdb
      POSTGRES_USER: user
      POSTGRES_HOST_AUTH_METHOD: trust

  chroma:
    image: chromadb/chroma:latest
    container_name: chroma_bench
    deploy:
      resources:
        limits:
          cpus: "6"
          memory: 4g
        reservations:
          cpus: "2"
          memory: 2g
    volumes:
      - chroma_data:/data
    environment:
      - IS_PERSISTENT=TRUE
      - ANONYMIZED_TELEMETRY=FALSE
```

The resource constraints ensure neither system gains unfair advantage from available hardware. Both containers receive:

- Maximum 6 CPU cores
- Maximum 4 GB memory
- Minimum guaranteed 2 CPU cores and 2 GB memory

D. Ensuring Fair Comparison

Following established benchmarking practices [**?**], several measures ensure benchmark fairness:

1) Database Reset Between Runs: Before each benchmark run, we reset both databases to clean state:

```bash
# Reset script clears all data
docker compose down -v
docker compose up -d
# Wait for services to be ready
sleep 10
```

This reset procedure removes Docker volumes, clearing all persisted data and forcing fresh initialization. For query benchmarks, we restart containers between ingestion and querying to clear in-memory caches (Chroma's HNSW graph cache and PostgreSQL's buffer pool). Note that this does not clear the host OS page cache, which may still contain recently-accessed data on Docker Desktop; however, since both systems run under identical conditions, this affects them equally.

2) Warmup Queries: For query benchmarks, we execute 10 warmup queries before measurements to eliminate cold-start effects from caching and JIT compilation.

3) Identical Datasets: Both systems ingest identical vector data generated with fixed random seeds for reproducibility.

E. Data Generation

Our data generator creates synthetic datasets with Gaussian distribution with controllable parameters for size, dimensionality and classes:

Listing 2: Data Generation

```python
class DatasetGenerator:
    def __init__(self, seed=42):
        np.random.seed(seed)

    def generate_vectors(self, size, dim):
        # Gaussian distribution
        return np.random.randn(size, dim)
            .astype(np.float32)

    def generate_metadata(self, size, classes,
                          class_distribution):
        sampled = np.random.choice(
            classes, size=size,
            p=class_distribution)
        return [{"cls": c} for c in sampled]
```

F. Dataset Configurations

The datasets used for benchmarking cover various scales and dimensionalities:

TABLE III: Dataset Configurations

Name	Vectors	Dimensions	Size (MB)
10k	10,000	128	5
50k	50,000	128	25
100k	100,000	128	50
500k	500,000	128	250
1m	1,000,000	128	500
2m	2,000,000	128	1,000
50k_32d	50,000	32	6
100k_32d	100,000	32	12
100k_768d	100,000	768	300
50k_1536d	50,000	1,536	300

Each vector has associated metadata with class labels (A, B, C) distributed according to configurable proportions (default: 10%, 30%, 60%).

G. Metrics Collection

During benchmarks, we collect:

- **Duration**: Wall-clock time for operations
- **Throughput**: Vectors per second (ingestion) or queries per second
- **Latency**: Mean, P50, P95, P99 query times
- **Recall**: Fraction of true nearest neighbors found
- **Docker Stats**: CPU and memory usage sampled during operations
- **Storage**: Disk usage before and after indexing

H. Recall Computation

Recall measures result quality against ground truth. Ground truth is computed per query using exact L2 search over the full dataset.

Listing 3: Recall Computation

```python
def compute_recall(results, ground_truth, k):
    gt_set = set(ground_truth[:k])
    result_set = set(results[:k])
    return len(gt_set & result_set) / k
```

I. Result Storage

All results are stored in JSON format for analysis:

```
{
  "experiment": "ingestion_benchmark",
  "environment": {
    "chroma_mem_limit": "4g",
    "pgvector_mem_limit": "4g",
    "cpus_limit": "6"
  },
  "runs": [
    {
      "dataset": "100k",
      "vectors": 100000,
      "dimensions": 128,
      "chroma": [...],
      "pgvector": [...]
    }
  ]
}
```

J. Result Processing and Visualization

The raw JSON results are processed by dedicated plotting scripts that parse, filter, and aggregate the data for visualization. Each plotting script loads the corresponding results file and applies filters based on the analysis requirements.

For ingestion benchmarks, we filter results by:

- **Index type**: Separating IVFFlat and HNSW runs for fair comparison
- **Batch size**: Selecting a consistent batch size (typically 1000) for throughput comparisons, or comparing across batch sizes for batch impact analysis
- **Dataset dimensions**: Grouping standard 128-dimensional datasets separately from variable-dimension experiments

For query benchmarks, filtering criteria include:

- **Filter mode**: Separating "nofilter" and "filter" query results
- **Top-K value**: Grouping by K=10, 50, or 100 for consistent comparisons
- **Index configuration**: Matching HNSW parameters (M, ef) or IVFFlat lists

The plotting scripts use pandas for data manipulation and matplotlib/seaborn for visualization. Results are aggregated across multiple runs where available, computing means and standard deviations for error bars. Plots are saved as PNG files with consistent styling for inclusion in this report.

An example plotting function for throughput comparison:

Listing 4: Example Plotting Function

```python
def plot_throughput_comparison(results_file):
    with open(results_file) as f:
        data = json.load(f)

    # Filter for standard datasets, batch=1000
    runs = [r for r in data['runs']
            if r['dimensions'] == 128]

    datasets = []
    chroma_throughput = []
    pgvector_throughput = []

    for run in runs:
        datasets.append(run['dataset'])
        # Find batch_size=1000 results
        chroma_run = next(
            r for r in run['chroma']
            if r['batch_size'] == 1000)
        pg_run = next(
            r for r in run['pgvector']
            if r['batch_size'] == 1000)
        chroma_throughput.append(
            chroma_run['vectors_per_second'])
        pgvector_throughput.append(
            pg_run['vectors_per_second'])

    # Create grouped bar chart
    x = np.arange(len(datasets))
    width = 0.35

    fig, ax = plt.subplots(figsize=(10, 6))
    ax.bar(x - width/2, chroma_throughput,
           width, label='Chroma')
    ax.bar(x + width/2, pgvector_throughput,
           width, label='pgvector')

    ax.set_ylabel('Vectors/second')
    ax.set_xlabel('Dataset')
    ax.set_xticks(x)
    ax.set_xticklabels(datasets)
    ax.legend()

    plt.savefig('throughput_comparison.png')
```

IV. Ingestion Benchmarks

A. Methodology

Ingestion benchmarks measure the time to load vectors into each database. For each dataset, we:

1) Reset containers to clean state
2) Load vectors in batches (500, 1000, 5000)
3) Create appropriate indexes
4) Record timing and resource usage

1) Chroma Ingestion: Chroma builds HNSW index incrementally during insertion:

Listing 5: Chroma Ingestion

```python
client = chromadb.HttpClient(
    host="localhost", port=8000)
collection = client.get_or_create_collection(
    name=collection_name,
    metadata={"hnsw:M": hnsw_m,
              "hnsw:construction_ef": ef})

for batch in batches:
    collection.add(
        ids=batch_ids,
        embeddings=batch_vectors.tolist(),
        metadatas=batch_metadata
    )
```

2) pgvector Ingestion: pgvector separates data insertion from index creation:

Listing 6: pgvector Ingestion

```
# Phase 1: Insert data
cur.executemany(
    "INSERT INTO vectors (embedding, cls) "
    "VALUES (%s, %s)",
    batch_data)
conn.commit()

# Phase 2: Create index
cur.execute(
"""CREATE INDEX ON vectors USING ivfflat (embedding
    vector_l2_ops) WITH (lists = 100)""")
```

B. Benchmark Phases

We conducted ingestion benchmarks in five phases:

Phase 1: Size Scaling (IVFFlat)

- Datasets: 10k, 50k, 100k, 500k, 1m, 2m
- Index: IVFFlat with lists=100
- Batch sizes: 500, 1000, 5000

Phase 2: Dimensionality Impact

- Datasets: 50k_32d, 100k_32d, 50k_1536d, 100k_768d
- Tests impact of vector dimensionality

Phase 3: HNSW Parameter Study

- Dataset: 50k
- Parameters: M $\in$ {16, 32}, ef $\in$ {64, 128}

Phase 4: IVFFlat Parameter Study

- Dataset: 50k
- Lists: 100, 200

Phase 5: HNSW at Scale

- Datasets: 10k, 100k, 500k, 1m
- HNSW with M=16, ef=64

C. Ingestion Results

1) Throughput Comparison: Table IV shows ingestion performance at batch size 1000:

TABLE IV: Ingestion Performance (batch_size=1000, IVFFlat)

Dataset	Chroma		pgvector	
	Time (s)	V/s	Time (s)	V/s
50k	7.81	6,405	13.34	3,840
100k	17.02	5,874	27.79	3,665
500k	117.10	4,270	133.14	3,814
1m	298.51	3,350	265.63	3,817

Key observations:

- Chroma achieves 1.5-1.7x higher throughput for smaller datasets
- Throughput advantage decreases as dataset size grows
- For 1M+ vectors, pgvector's throughput remains stable while Chroma's drops

2) Batch Size Impact: Table V shows how batch size affects throughput:

TABLE V: Batch Size Impact on Throughput (500k dataset)

System	500	1000	5000
Chroma (V/s)	4,238	4,270	5,569
pgvector (V/s)	3,884	3,814	3,838

Chroma benefits significantly from larger batches (31% improvement), while pgvector shows minimal change. This reflects Chroma's HTTP overhead being amortized over larger batches.

TABLE VI: Storage Requirements (MB)

Dataset	Chroma	pgvector	pg (pre-index)
50k	33.1	55.5	29.0
100k	66.2	110.6	58.0
500k	331.9	550.7	289.9
1m	664.0	1,099.1	579.3

3) Storage Comparison: Chroma uses approximately 40% less storage than pgvector with IVFFlat index. pgvector's IVFFlat index nearly doubles raw data size.

TABLE VII: Peak Memory Usage During Ingestion

Dataset	Chroma	pgvector
50k	132 MB (3.2%)	88 MB (2.1%)
100k	240 MB (5.9%)	149 MB (3.6%)
500k	1.07 GB (26.6%)	275 MB (6.7%)
1m	2.06 GB (51.6%)	401 MB (9.8%)

4) Memory Usage: Chroma's memory usage grows significantly with dataset size due to HNSW graph construction, while pgvector maintains relatively flat memory consumption.

5) CPU Utilization: Table VIII shows average CPU utilization during ingestion (batch size = 1000). Note that CPU percentages can exceed 100% when multiple cores are utilized.

TABLE VIII: Average CPU Utilization During Ingestion (%)

Dataset	Chroma	pgvector
50k	159.7	32.4
100k	150.4	35.6
500k	178.8	34.8
1m	174.7	33.0

Chroma consistently utilizes 4-5x more CPU than pgvector during ingestion. This reflects Chroma's parallel HNSW graph construction, which builds index structures during insertion. pgvector's lower CPU usage corresponds to its simpler insertion path. Data is written sequentially, with index construction deferred to a separate phase. Despite higher CPU utilization, Chroma achieves better throughput, indicating efficient parallelization of the indexing workload.

6) Index Build Time: pgvector reports separate timings for data insertion and index creation:

TABLE IX: pgvector Time Breakdown (IVFFlat, batch=1000)

Dataset	Insert (s)	Index (s)	Total (s)
50k	13.02	0.31	13.34
100k	27.29	0.51	27.79
500k	131.10	2.04	133.14
1m	261.17	4.46	265.63

IVFFlat index creation is remarkably fast (1.5-1.7% of total time). Chroma's HNSW is built incrementally during insertion, so no separate timing is available.

D. Ingestion Visualizations

Figure 1 shows the ingestion throughput comparison across dataset sizes, ignoring the index build time (only for pgvector, chroma index building is built-in).

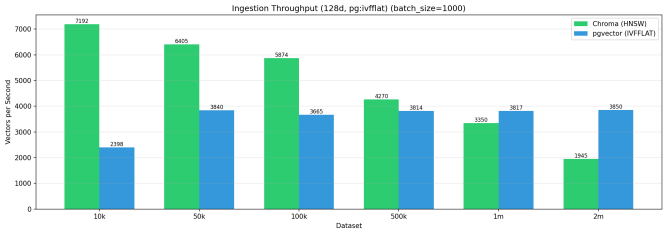

Fig. 1: Ingestion Throughput Comparison

Figure 2 shows the total ingestion time breakdown, illustrating how time is distributed between data insertion and index creation, using IVFFlat.

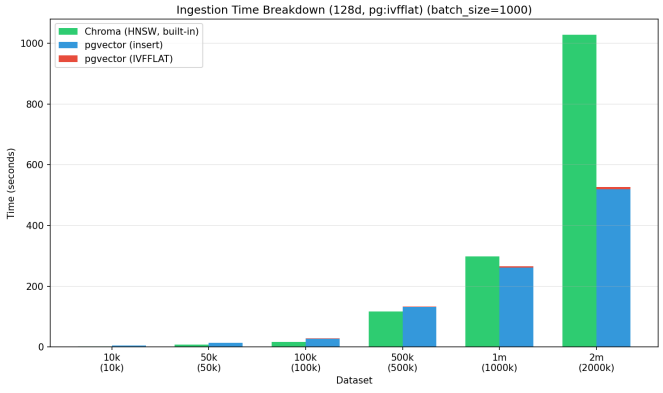

Fig. 2: Total Ingestion Time Breakdown (IVFFlat)

Figure 3 shows the same statistic using HNSW index for pgvector (fair comparison for the two databases). The overhead introduced for building this index is massive in comparison to IVFFlat.

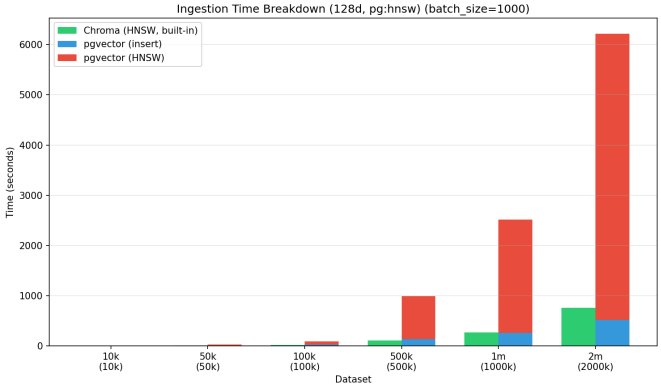

Fig. 3: Total Ingestion Time Breakdown (HNSW)

Figure 4 illustrates memory usage during ingestion, showing Chroma's higher memory consumption.

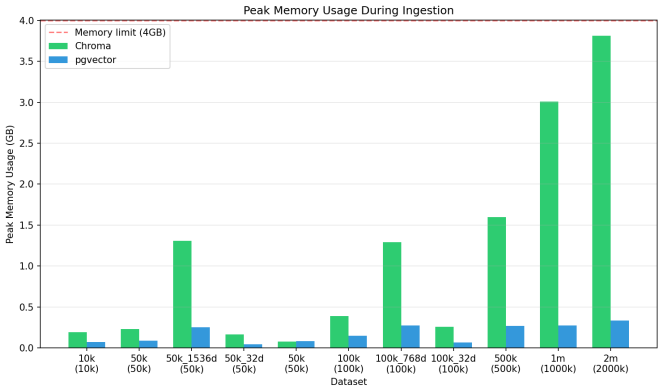

Fig. 4: Memory Usage During Ingestion

Figure 5 compares storage requirements between the two systems.

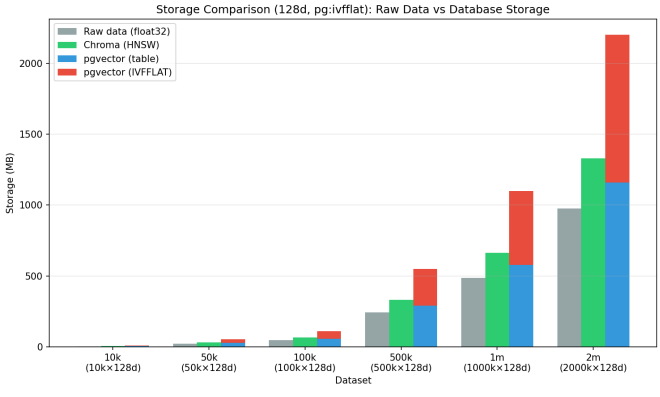

Fig. 5: Storage Comparison

1) Dimensionality Impact: Vector dimensionality significantly affects ingestion performance. Table X shows throughput and storage for datasets with dimensions ranging from 32 to 1536 (batch size = 1000).

TABLE X: Dimensionality Impact on Ingestion (batch=1000)

Dataset	Dim	Chroma V/s	Chroma MB	pgvector V/s	pgvector MB
50k_32d	32	7,747	14.8	6,175	17.0
50k	128	6,405	33.1	3,840	55.5
50k_1536d	1,536	1,583	301.7	259	790.1
100k_32d	32	7,050	29.6	6,206	33.4
100k	128	5,874	66.2	3,665	110.6
100k_768d	768	2,566	310.4	314	793.4

Key observations:

- Both systems show throughput degradation with higher dimensions
- Chroma maintains 6x higher throughput than pgvector at high dimensions (1536d)
- pgvector's throughput drops dramatically (24x slower at 768d vs 32d)
- Storage scales linearly with dimensionality for both systems
- pgvector requires 2-2.5x more storage than Chroma across all dimensions

E. Index Analysis

A key differentiator between pgvector and Chroma is how indexes are constructed. pgvector separates data insertion from index creation, allowing us to measure each phase independently. Chroma builds its HNSW index incrementally during insertion, making these phases inseparable.

Figure 6 compares index build times for pgvector's IVFFlat and HNSW indexes across dataset sizes.

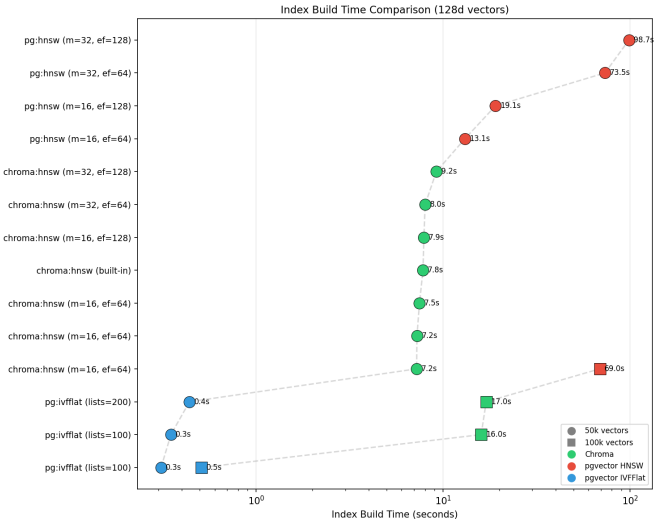

Fig. 6: Index Build Time Comparison (pgvector)

Key observations from the index analysis:

- **IVFFlat**: Extremely fast index creation (under 2% of total time), making it ideal for bulk loading scenarios
- **HNSW**: Slower to build but provides better query performance; build time scales with dataset size
- **Trade-off**: IVFFlat offers faster ingestion while HNSW offers better recall at query time

Figure 7 shows the storage overhead introduced by the IVFFlat index.

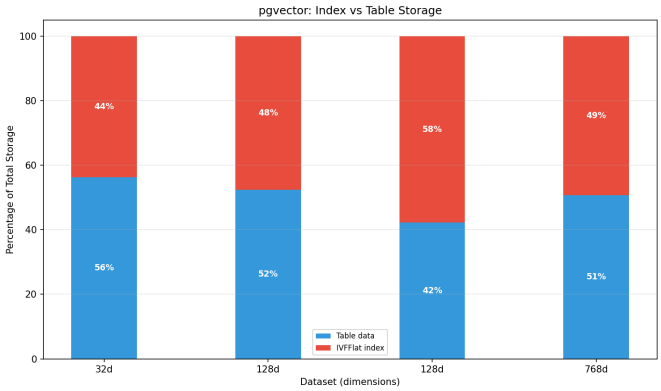

Fig. 7: Index Storage Overhead

The storage analysis reveals that IVFFlat indexes add significant overhead (nearly doubling raw data size), while HNSW indexes in Chroma maintain a more compact representation due to the graph-based structure storing only neighbor pointers rather than cluster centroids.

F. Ingestion Analysis

Chroma Strengths:

- Higher throughput for datasets under 500k vectors
- More compact storage due to HNSW index structure
- Simpler API with automatic index management

Chroma Weaknesses:

- Memory consumption grows substantially with dataset size
- Throughput degrades for large datasets (1M+)

pgvector Strengths:

- Consistent throughput regardless of dataset size
- Lower, predictable memory usage
- Fast IVFFlat index creation

pgvector Weaknesses:

- Lower absolute throughput than Chroma
- Higher storage overhead with IVFFlat index
- Very slow HNSW index creation

V. Query Benchmarks

A. Methodology

Query benchmarks measure search performance. For each configuration:

1) Ingest dataset with specified index
2) Restart containers to clear caches
3) Compute ground truth via brute-force
4) Execute 10 warmup queries
5) Run 100 query iterations per top-K value
6) Record latency and recall statistics

We test top-K values of 10, 50, and 100, both with and without metadata filters.

B. Query Implementation

```
# Without filter
results = collection.query(
    query_embeddings=[query_vector],
    n_results=k)

# With filter
results = collection.query(
    query_embeddings=[query_vector],
    n_results=k,
    where={"cls": filter_class})
```

Listing 7: pgvector Query

```
-- Without filter (L2 distance)
SELECT id FROM vectors
ORDER BY embedding <-> %s
LIMIT %s;

-- With filter
SELECT id FROM vectors
WHERE cls = %s
ORDER BY embedding <-> %s
LIMIT %s;
```

C. Benchmark Phases

Phase 1: Size Scaling

- Datasets: 10k through 2m
- Index: HNSW (M=16, ef=64)
- Modes: with and without filter

Phase 2: Index Comparison

- Dataset: 50k
- Indexes: IVFFlat (lists=100, 200), HNSW (various M/ef)

D. Query Results

1) Latency Comparison: Table XI shows query latency for HNSW index:

TABLE XI: Query Latency (ms) - HNSW, k=10, no filter

Dataset	Chroma				pgvector			
	Mean	P50	P95	P99	Mean	P50	P95	P99
10k	1.76	1.75	1.94	1.99	0.83	0.75	1.26	1.50
50k	1.65	1.59	1.97	2.11	1.03	0.93	1.49	2.20
100k	1.54	1.52	1.70	1.95	1.22	1.06	1.99	2.65
500k	1.81	1.62	2.35	2.89	2.27	2.11	3.41	4.21

pgvector consistently achieves 2x lower latency than Chroma for unfiltered queries. Further results indicate that Mean, P50, P95 and P99 metrics follow similar patterns in all test scenarios so we showcase only the Mean metric in the following experiments.

2) Recall Comparison: Table XII shows recall accuracy:

TABLE XII: Recall@10 - HNSW, no filter

Dataset	Chroma		pgvector	
	Mean	Min	Mean	Min
10k	0.916	0.60	0.868	0.40
50k	0.732	0.30	0.795	0.48
100k	0.689	0.28	0.742	0.43

Chroma shows higher recall on small datasets, while pgvector performs better as size increases.

3) Filter Impact: Metadata filtering affects both systems:

TABLE XIII: Filter Impact on Latency (50k, HNSW, k=10)

Mode	Chroma		pgvector	
	Mean (ms)	Recall	Mean (ms)	Recall
No filter	1.65	0.732	0.89	0.795
With filter	3.81	0.624	0.84	0.585

Key observations:

- Chroma's latency increases 2.3x with filters
- pgvector's latency remains nearly unchanged
- Both systems show recall degradation with filters

4) Top-K Impact: Larger K values affect performance:

TABLE XIV: Top-K Impact (50k, HNSW, no filter)

k	Chroma		pgvector	
	Latency (ms)	Recall	Latency (ms)	Recall
10	1.65	0.732	0.89	0.795
50	2.52	0.615	0.89	0.796
100	2.60	0.557	0.70	0.796

Chroma shows latency increase with K while pgvector remains stable. Recall patterns differ between systems.

E. Query Visualizations

1) Latency Analysis: Figure 8 shows the query latency comparison across dataset sizes for k=10.

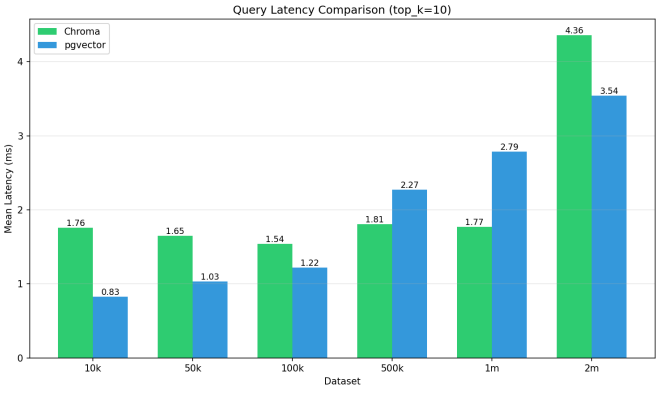

Fig. 8: Query Latency Comparison (k=10, no filter)

2) Recall Analysis: Figure 9 compares recall accuracy between the systems for k=10.

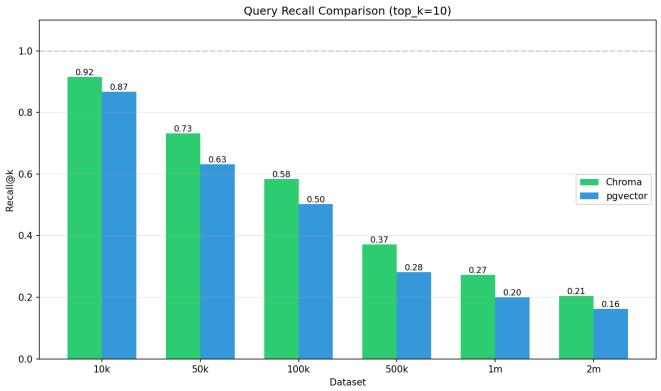

Fig. 9: Recall Comparison (k=10, no filter)

Figure 10 shows the latency-recall tradeoff, illustrating the fundamental tension between query speed and result quality.

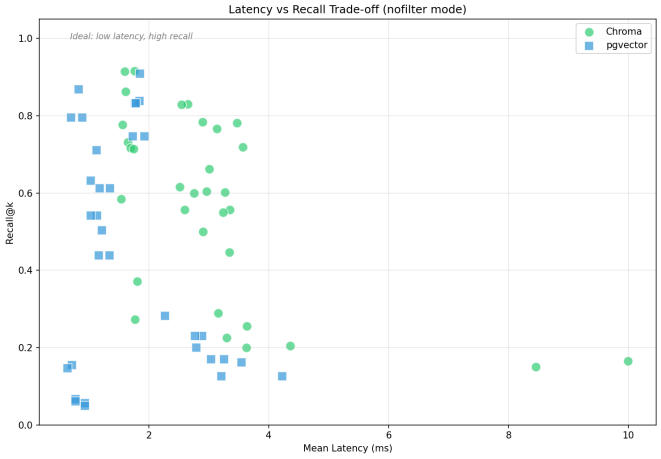

Fig. 10: Latency vs Recall Tradeoff

3) Scalability Analysis: Figure 11 shows how latency scales with dataset size, demonstrating both systems' ability to handle increasing data volumes.

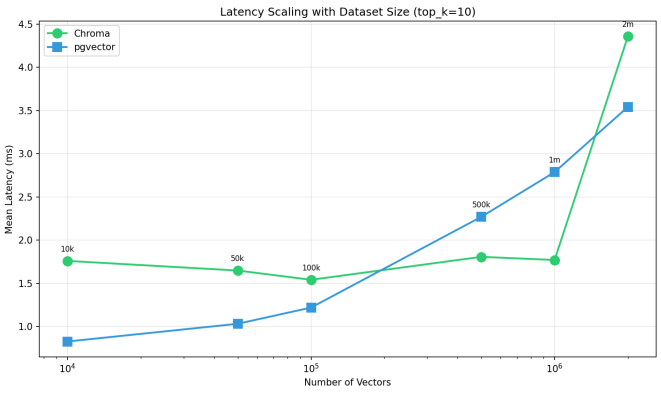

Fig. 11: Latency Scaling with Dataset Size

Figure 12 shows recall scaling behavior across dataset sizes.

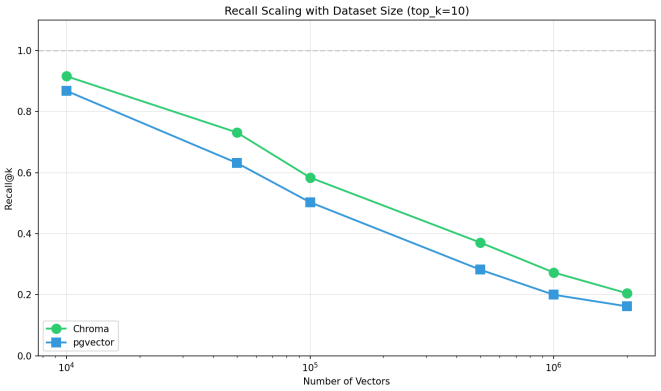

Fig. 12: Recall Scaling with Dataset Size

4) Filter Impact Analysis: Metadata filtering is a common operation in production systems. Figure 13 shows how filtering affects query latency.

Fig. 13: Filter Impact on Query Latency

Figure 14 shows the recall degradation when metadata filters are applied.

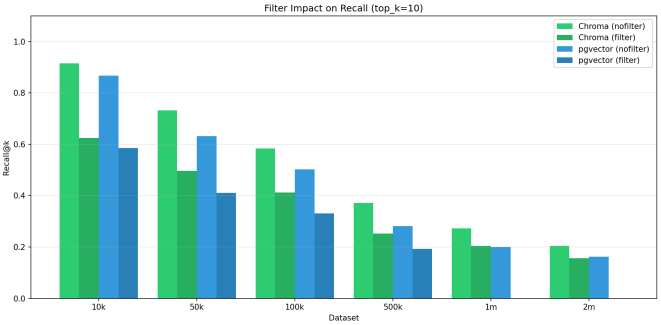

Fig. 14: Filter Impact on Recall

Key observations from filter analysis:

- Chroma's latency increases significantly with filters due to post-filtering approach
- pgvector maintains nearly constant latency thanks to SQL WHERE clause optimization
- Both systems show recall degradation with filters, but pgvector degrades more gracefully

Note on filter implementation differences: The performance gap for filtered queries reflects a fundamental architectural difference, not a benchmarking artifact. Chroma performs

post-filtering: it first retrieves candidates via HNSW traversal, then filters by metadata, potentially requiring multiple traversal rounds to find enough matching results. pgvector integrates filtering into query planning via SQL WHERE clauses, allowing the query optimizer to potentially use metadata indexes. This is a system-level design choice with legitimate trade-offs. Chroma's approach is simpler to implement and may perform better when filters are highly selective, while pgvector's approach excels when filters match a larger fraction of the dataset.

5) Top-K Impact Analysis: Table XV presents a comprehensive view of how Top-K values affect both latency and recall across all dataset sizes.

TABLE XV: Top-K Impact on Latency (ms) and Recall - HNSW, No Filter

Dataset	Metric	Chroma k=10	Chroma k=50	Chroma k=100	pgvector k=10	pgvector k=50	pgvector k=100
10k	Latency	1.76	2.65	2.89	0.83	0.89	0.70
	Recall	0.92	0.83	0.78	0.87	0.80	0.80
50k	Latency	1.65	2.52	2.60	1.03	1.13	1.03
	Recall	0.73	0.62	0.56	0.63	0.54	0.54
100k	Latency	1.54	2.90	3.34	1.22	1.34	1.16
	Recall	0.58	0.50	0.45	0.50	0.44	0.44
500k	Latency	1.81	3.16	3.64	2.27	2.89	2.77
	Recall	0.37	0.29	0.26	0.28	0.23	0.23
1m	Latency	1.77	3.30	3.63	2.79	3.25	3.04
	Recall	0.27	0.23	0.20	0.20	0.17	0.17
2m	Latency	4.36	9.99	8.45	3.54	4.22	3.21
	Recall	0.21	0.17	0.15	0.16	0.13	0.12

Key observations from the Top-K analysis:

- **Latency scaling**: Chroma's latency increases significantly with K (up to 2x from k=10 to k=100), while pgvector remains relatively stable
- **Recall degradation**: Both systems show recall decrease with larger K, but the effect is more pronounced in Chroma
- **System comparison**: pgvector maintains lower latency across all K values, especially for larger datasets
- **Dataset scaling**: Recall drops for both systems as dataset size increases, as expected for ANN algorithms

F. Index Parameter Analysis

A critical aspect of ANN search performance is the choice of index type and its parameters. We conducted detailed experiments on the 50k dataset to analyze how different index configurations affect query performance.

1) Index Configurations Tested: For pgvector, we tested:

- **HNSW**: M ∈ {16, 32}, ef_construction ∈ {64, 128}
- **IVFFlat**: lists ∈ {100, 200}

Chroma uses HNSW exclusively with its default parameters.

2) Index Parameter Impact on Latency: Figure 15 shows how different index configurations affect query latency.

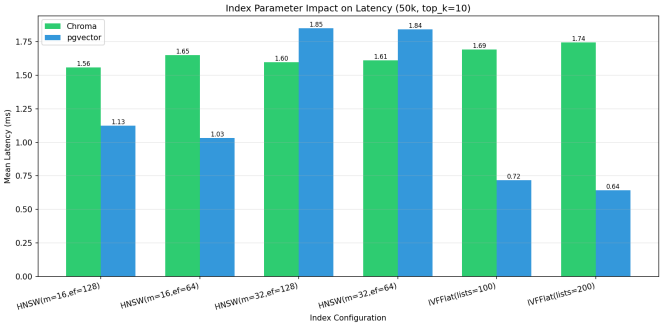

Fig. 15: Index Parameter Impact on Query Latency (50k dataset)

Key observations:

- IVFFlat with more lists (200) provides faster queries than fewer lists (100)
- HNSW parameters have moderate impact on latency
- Higher M values in HNSW slightly increase latency due to more neighbor comparisons

3) Index Parameter Impact on Recall: Figure 16 shows recall accuracy across different index configurations.

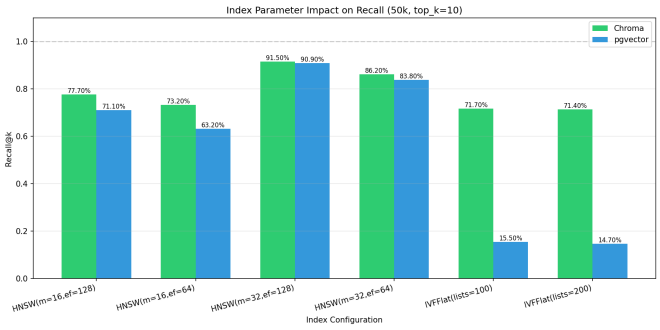

Fig. 16: Index Parameter Impact on Recall (50k dataset)

Key observations:

- HNSW generally achieves higher recall than IVFFlat
- Higher ef_construction values improve HNSW recall
- IVFFlat recall improves with fewer lists (100 vs 200) at the cost of latency

4) Latency-Recall Tradeoff by Index Configuration: Figure 17 visualizes the latency-recall tradeoff for each index configuration, helping identify optimal configurations for different use cases.

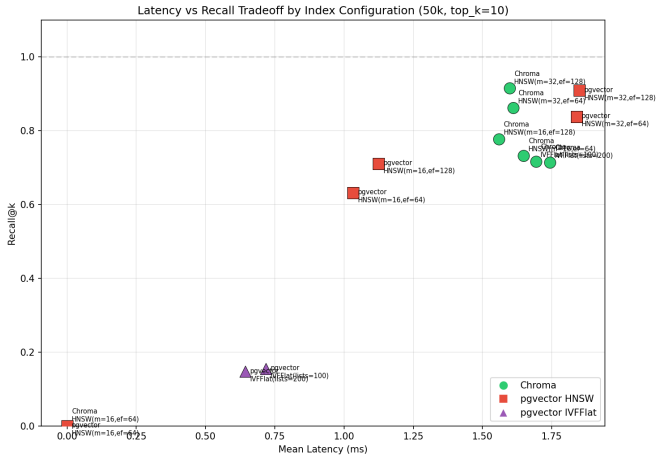

Fig. 17: Latency vs Recall Tradeoff by Index Configuration (50k dataset)

The ideal configuration lies in the top-left corner (low latency, high recall). This plot helps practitioners choose configurations based on their specific latency and accuracy requirements.

5) HNSW vs IVFFlat Comparison: Figure 18 provides a direct comparison between HNSW and IVFFlat index types.

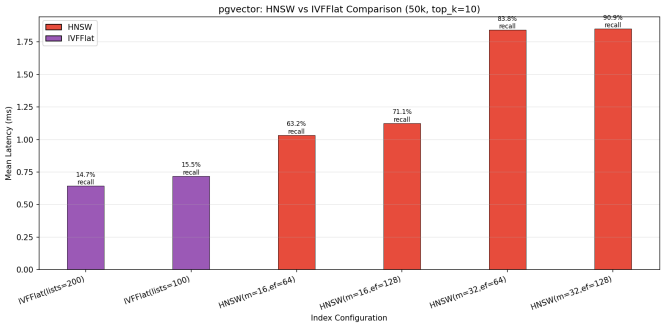

Fig. 18: HNSW vs IVFFlat Index Type Comparison (50k dataset)

Summary of index type characteristics:

- **HNSW**: Better recall, slightly higher latency, more memory during construction
- **IVFFlat**: Faster queries, lower recall, much faster index creation
- For applications prioritizing recall: Use HNSW with higher M and ef values
- For applications prioritizing throughput: Use IVFFlat with more lists

G. Resource Utilization During Queries

Tables XVI and XVII show hardware resource utilization during query execution (HNSW, k=10, no filter).

TABLE XVI: CPU Utilization During Queries (%)

Dataset	Chroma		pgvector	
	Avg	Peak	Avg	Peak
10k	14.04	14.04	0.04	0.04
50k	13.28	13.28	0.06	0.06
100k	12.34	12.34	0.06	0.06

TABLE XVII: Memory Utilization During Queries

Dataset	Chroma		pgvector	
	% of 4GiB	Peak Usage	% of 4GiB	Peak Usage
10k	0.69	28.16 MiB	0.75	30.9 MiB
50k	1.67	68.29 MiB	1.58	64.64 MiB
100k	2.85	116.6 MiB	2.54	103.9 MiB
500k	11.83	484.7 MiB	3.57	146.4 MiB

Query resource utilization differs dramatically between systems. Chroma shows moderate CPU usage (12–14%) for smaller datasets during graph traversal. pgvector maintains near-zero CPU overhead during queries, reflecting PostgreSQL's efficient query execution model, queries complete so quickly that CPU sampling barely registers activity.

Note on CPU sampling methodology: CPU utilization was measured via Docker's `stats` API, which samples at approximately 1-second intervals. For short query batches (100 queries completing in under 1 second), this sampling granularity may underreport CPU usage, particularly for pgvector where individual queries complete in sub-millisecond time. The near-zero values for pgvector in Table XVI reflect this limitation rather than zero actual CPU usage.

Memory consumption reveals a critical scalability difference. Both systems show comparable memory footprints for smaller datasets (10k–100k vectors), with Chroma slightly higher due to its in-memory graph structure. However, at 500k vectors, the divergence becomes pronounced: Chroma requires 484.7 MiB (11.83% of the 4 GiB container limit), while pgvector maintains a modest 146.4 MiB footprint (3.57%). This 3.3× difference reflects Chroma's requirement to keep the full HNSW graph in memory, whereas pgvector leverages PostgreSQL's buffer pool and can page index segments from disk as needed.

H. Sustained Query Workload Analysis

To understand resource utilization under sustained load, we also measured performance with 1000 queries (HNSW, k=50, no filter). Table XVIII shows the results.

TABLE XVIII: CPU Utilization During Sustained Queries (%) - k=50, 1000 queries

Dataset	Chroma		pgvector	
	Avg	Peak	Avg	Peak
100k	54.59	54.82	44.42	44.42

With a sustained workload of 1000 queries, both systems show substantial CPU utilization. Chroma averages 54.6%

CPU while pgvector uses 44.4%, a 23% difference that indicates pgvector's more efficient query execution path. This contrasts with near-zero utilization seen in smaller query batches, revealing that sustained load exposes the true computational cost of HNSW traversal.

Chroma's memory-bound architecture provides consistent latency but requires careful capacity planning, particularly for datasets exceeding millions of vectors. pgvector's lower memory overhead and lower CPU utilization make it attractive for resource-constrained environments or when vector search is one of many concurrent database workloads.

I. Query Analysis Summary

Latency: pgvector achieves consistently lower query latency (approximately 2x faster), likely due to its native SQL execution engine and efficient index implementation. The latency advantage holds across all dataset sizes and configurations tested.

Recall: Results vary by configuration with no clear winner. Chroma shows higher recall on smaller datasets, while pgvector performs better as size increases. Index parameter tuning significantly affects recall for both systems.

Filters: pgvector handles metadata filters more efficiently, with minimal latency impact compared to Chroma's 2-3x slowdown. This makes pgvector more suitable for applications requiring filtered vector search.

Scalability: Both systems show graceful degradation as dataset size increases, with pgvector maintaining better latency characteristics at scale.

Index Selection: HNSW provides better recall at the cost of slower index creation and slightly higher query latency. IVFFlat offers faster index creation and queries but with lower recall. The choice depends on whether the application prioritizes accuracy or throughput.

VI. Limitations and Future Work

A. Limitations

Our study has several limitations:

1) **Single-node only**: Distributed configurations not evaluated
2) **Synthetic data**: Random Gaussian vectors may not represent real embedding distributions; future work could use standard benchmarks like SIFT1M [17]
3) **Cold-start focus**: Warm cache scenarios not extensively tested

B. Future Work

The following could be done to further support our conclusions:

- **Real embeddings**: Testing with actual ML model outputs
- **Concurrent workloads**: Multi-client performance testing
- **Update operations**: Measuring insert/delete/update performance
- **Memory constraints**: Performance under memory pressure
- **Further experimentation with parameters**: More parameters for indexes, sizes, dimensionality could be examined

VII. Conclusion

This report presented a comprehensive performance comparison between Chroma and pgvector for vector similarity search workloads. Our key findings:

Ingestion Performance:

- Chroma achieves 1.5-1.7x higher throughput for small-to-medium datasets
- pgvector maintains consistent throughput at scale
- Chroma uses less storage but more memory
- pgvector's IVFFlat index creation is very fast (under 2% of total time)

Query Performance:

- pgvector achieves 2x lower query latency
- Recall varies by configuration with no clear winner
- pgvector handles metadata filters more efficiently
- Both systems scale reasonably to million-vector datasets

Recommendations:

Choose **Chroma** when:

- Building prototypes or RAG applications
- Dataset size under 500k vectors
- Operational simplicity is prioritized

Choose **pgvector** when:

- Integrating with existing PostgreSQL infrastructure
- Low latency is critical requirement
- Handling million+ vector datasets
- Complex queries combining relational and vector operations
- ACID guarantees are required

All benchmarking code, datasets, and results are available in our repository. A detailed README explains the complete setup and reproduction process.

https://github.com/anagnole/vec3-comparison
https://github.com/users/anagnole/projects/1

Acknowledgment of AI Assistance

Generative AI tools were used during this project for productivity and writing support, as summarized below.

- **Background research and reading guidance:** ChatGPT [7] was used to propose candidate reading materials and keywords relevant to vector databases, similarity search, and the technologies used in this study.
- **Planning and organization:** ChatGPT was used to help outline an implementation plan and a tentative project timeline.
- **Code assistance (implementation and debugging):** Microsoft Copilot [8](Agent mode) was used to assist in implementing plotting utilities, automation bash script creation and the web interface on top of the existing project infrastructure. Agent mode was used by providing explicit user instructions and requirements for each plot. For example, after the benchmark results were generated in the predefined JSON format, Copilot was prompted to produce a plotting function that loads the results, filters them by factors such as database system, dimensionality,

and index type, and visualizes the effect of index configuration on memory usage using a bar chart. Copilot was also used to assist with debugging and error handling.
- **Documentation:** Microsoft Copilot was used to assist in drafting the repository `README`.
- **Report:** Given a draft version of the report (can be found in the repository), ChatGPT expnaded on bullet point information to assist in writing the full report. The initial draft report included instructions, results, structural information and interpretation of results. In addition to that, AI assisted with:
  - Language editing and rephrasing of selected paragraphs.
  - Suggestions on formatting and wording.

All experimental design and benchmarking decisions, measurements, plots, code organization, repository setup and version control, tool selection, hardware specifications, and result interpretations were produced by the author.

References

[1] Chroma, "Chroma - the AI-native open-source embedding database," 2023. [Online]. Available: https://www.trychroma.com/

[2] A. Kane, "pgvector: Open-source vector similarity search for Postgres," 2023. [Online]. Available: https://github.com/pgvector/pgvector

[3] M. Aumüller, E. Bernhardsson, and A. Faithfull, "ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms," *Information Systems*, vol. 87, p. 101374, 2020.

[4] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 42, no. 4, pp. 824–836, 2020.

[5] H. Jégou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 33, no. 1, pp. 117–128, 2011.

[6] PostgreSQL Global Development Group, "PostgreSQL: The World's Most Advanced Open Source Relational Database," 2023. [Online]. Available: https://www.postgresql.org/

[7] OpenAI, "Introducing ChatGPT," Nov. 30, 2022. [Online]. Available: https://openai.com/index/chatgpt/

[8] Microsoft, "Introducing GitHub Copilot agent mode (preview)," Feb. 24, 2025. [Online]. Available: https://code.visualstudio.com/blogs/2025/02/24/introducing-copilot-agent-mode

[9] Pinecone, "What is a Vector Database?" 2023. [Online]. Available: https://www.pinecone.io/learn/vector-database/

[10] Pinecone, "Approximate Nearest Neighbor (ANN) Search," 2023. [Online]. Available: https://www.pinecone.io/learn/ann/

[11] Qdrant, "Distance Metrics," 2023. [Online]. Available: https://qdrant.tech/documentation/concepts/distance/

[12] Milvus, "Index Selection," 2023. [Online]. Available: https://milvus.io/docs/index_selection.md

[13] Qdrant, "Vector Database Benchmark," 2023. [Online]. Available: https://github.com/qdrant/vector-db-benchmark

[14] Zilliz, "Benchmarking Vector Databases," 2023. [Online]. Available: https://zilliz.com/blog/benchmarking-vector-databases

[15] Supabase, "Getting Started with pgvector," 2023. [Online]. Available: https://supabase.com/blog/pgvector-getting-started

[16] Supabase, "pgvector Performance," 2023. [Online]. Available: https://supabase.com/blog/pgvector-performance

[17] INRIA, "TEXMEX: Multimedia Indexing and Search," 2023. [Online]. Available: http://corpus-texmex.irisa.fr/

[18] LangChain, "Chroma Vector Store Integration," 2023. [Online]. Available: https://python.langchain.com/docs/integrations/vectorstores/chroma

Appendix A
Complete Setup Guide

This appendix provides step-by-step instructions to reproduce our benchmarks.

A. Prerequisites

- Docker Desktop (with Docker Compose v2)
- Python 3.9 or later
- Git
- At least 8GB RAM and 10GB disk space

B. Repository Setup

```
# Clone the repository
git clone https://github.com/anagnole/vec3-
    comparison
cd vec3-comparison

# Create Python virtual environment
python3 -m venv .venv
source .venv/bin/activate

# Install dependencies
pip install -r requirements.txt
```

C. Docker Database Setup

Start both database containers:

```
# Start containers
docker compose up -d

# Verify containers are running
docker compose ps

# Expected output:
# chroma_bench    running  0.0.0.0:8000->8000
# pgvector_bench  running  0.0.0.0:5432->5432
```

D. Smoke Tests

Verify both databases are accessible:

```
# Test Chroma connectivity
python -c "
import␣chromadb
client␣=␣chromadb.HttpClient(
␣␣␣␣host='localhost',␣port=8000)
print('Chroma␣heartbeat:',␣client.heartbeat())
"

# Test pgvector connectivity
python -c "
import␣psycopg2
conn␣=␣psycopg2.connect(
␣␣␣␣dbname='vecdb',␣user='user',
␣␣␣␣host='localhost',␣port='5432')
cur␣=␣conn.cursor()
cur.execute('SELECT␣version()')
print('PostgreSQL:',␣cur.fetchone()[0][:50])
"
```

E. Data Generation

Generate test datasets:

```
# Small test dataset
python vec3/generate_data.py \
    --size 10000 --dim 128 \
    --out data/10k

# Medium dataset
python vec3/generate_data.py \
    --size 100000 --dim 128 \
    --out data/100k
```

```
# Large dataset
python vec3/generate_data.py \
    --size 500000 --dim 128 \
    --out data/500k

# High-dimensional dataset
python vec3/generate_data.py \
    --size 100000 --dim 768 \
    --out data/100k_768d
```

F. Database Reset Scripts

Before each benchmark, reset databases:

```
./scripts/db/reset_and_wait.sh

./scripts/db/restart.sh
```

G. Running Ingestion Benchmarks

Single dataset:

```
.venv/bin/python \
    benchmarks/ingestion/run_single_dataset.py \
    -d 100k -i ivfflat --lists 100
```

Full benchmark suite:

```
./scripts/benchmarks/run_all_ingestion.sh
```

H. Running Query Benchmarks

Single dataset:

```
.venv/bin/python \
    benchmarks/queries/run_single_dataset.py \
    100k --both -i hnsw
```

Full benchmark suite:

```
./scripts/benchmarks/run_all_queries.sh
```

I. Generating Plots

```
# Ingestion plots
.venv/bin/python \
    benchmarks/plotting/ingestion_plots.py

# Query plots
.venv/bin/python \
    benchmarks/plotting/queries_plots.py
```

Plots are saved to `results/plots/ingestion/` and `results/plots/queries/`.

J. Web Interface (Alternative)

For interactive benchmarking, use the web UI:

```
# Start API and web frontend
./scripts/ui/start.sh

# Access at http://localhost:5173
```

The web interface provides:

- Docker container management
- Dataset generation with parameters
- Interactive ingestion and query benchmarks
- Real-time log viewing
- Plot generation and visualization

Appendix B
Project Structure

```
vec3-comparison/
  docker-compose.yml     # Database containers
  requirements.txt       # Python dependencies

  vec3/                  # Core modules
    generate_data.py     # Data generator
    ingest_chroma.py     # Chroma ingestion
    ingest_pgvector.py   # pgvector ingestion
    query_chroma.py      # Chroma queries
    query_pgvector.py    # pgvector queries
    metrics.py           # Recall computation

  benchmarks/
    ingestion/           # Ingestion benchmarks
    queries/             # Query benchmarks
    plotting/            # Result visualization

  scripts/
    db/                  # Database management
    benchmarks/          # Benchmark runners
    ui/                  # Web UI scripts

  api/                   # Node.js API server
  web/                   # React frontend

  results/
    raw/                 # JSON results
    plots/               # Generated charts

  docs/
    reading_quide.md     # Related reading material
    protocol.md          # Implementation protocol
    scripts.md           # Scripts explanation
    charter.md           # Project charter
```

The repository README provides complete documentation for all scripts and options.