

In [ ]:

# Comparativa: almacenamiento y recuperación local de embeddings

Buscamos el equivalente a SQLite/DuckDB para vectores: **embebido, sin servidor, archivo local**.

Candidatos:

- **FAISS** — Meta, gold standard ANN, sin metadata nativa
- **sqlite-vec** — Extensión SQLite pura en C, vectores junto a datos SQL
- **LanceDB** — DB columnar para vectores, persiste en directorio
- **ChromaDB** — Popular, metadata rica, modo local
- **USearch** — Ultra ligero, rivaliza FAISS en velocidad

Medimos: inserción, búsqueda top-10, persistencia a disco, carga desde disco, tamaño en disco.

## 1. Setup y corpus de embeddings pre-generados

```
In [1]: import numpy as np
import time, os, shutil, json
import pandas as pd
import matplotlib.pyplot as plt

plt.style.use('seaborn-v0_8-whitegrid')

# Generamos embeddings sintéticos (dim=384 como e5-small) para no depender de
# Esto aísla el benchmark de storage del benchmark de encoding
np.random.seed(42)

SIZES = [1_000, 10_000, 50_000]
DIM = 384
DATA_DIR = 'data/vector_bench'
os.makedirs(DATA_DIR, exist_ok=True)

# Pre-generar embeddings normalizados (simulan output de e5-small)
datasets = {}
for n in SIZES:
    vecs = np.random.randn(n, DIM).astype(np.float32)
    # Normalizar como haría sentence-transformers
    norms = np.linalg.norm(vecs, axis=1, keepdims=True)
    vecs = vecs / norms
    datasets[n] = vecs
```

```

# Queries (100 vectores)
N_QUERIES = 100
queries = np.random.randn(N_QUERIES, DIM).astype(np.float32)
queries = queries / np.linalg.norm(queries, axis=1, keepdims=True)

# Metadata simulada
def make_metadata(n):
    categories = ['programación', 'ciencia', 'cocina', 'finanzas', 'geografía']
    return [{'id': i, 'category': categories[i % len(categories)], 'text': f'

print(f'Dimensión: {DIM}')
print(f'Tamaños de corpus: {SIZES}')
print(f'Queries: {N_QUERIES}')
print(f'Embeddings pre-generados (normalizados, float32)')

```

Dimensión: 384  
 Tamaños de corpus: [1000, 10000, 50000]  
 Queries: 100  
 Embeddings pre-generados (normalizados, float32)

## 2. Benchmark framework

Cada backend implementa: insert, search, save, load, size\_on\_disk.

```

In [2]: K = 10 # top-k para búsqueda

def bench_backend(name, insert_fn, search_fn, save_fn, load_search_fn, size_
    """Benchmark completo de un backend de vectores."""
    results = []
    for n in SIZES:
        vecs = datasets[n]
        meta = make_metadata(n)
        path = os.path.join(DATA_DIR, f'{name}_{n}')

        # Limpiar estado previo
        cleanup_fn(path)

        # INSERT
        t0 = time.perf_counter()
        state = insert_fn(vecs, meta, path)
        insert_time = time.perf_counter() - t0

        # SEARCH (100 queries)
        t0 = time.perf_counter()
        results_search = search_fn(state, queries, K)
        search_time = time.perf_counter() - t0
        search_per_query = search_time / N_QUERIES

        # SAVE / persist
        t0 = time.perf_counter()
        save_fn(state, path)
        save_time = time.perf_counter() - t0

        # SIZE on disk
        disk_mb = size_fn(path)

```

```

# LOAD from disk + search
t0 = time.perf_counter()
loaded_results = load_search_fn(path, queries[:1], K)
load_search_time = time.perf_counter() - t0

results.append({
    'backend': name,
    'n_vectors': n,
    'insert_s': round(insert_time, 4),
    'search_100q_s': round(search_time, 4),
    'per_query_ms': round(search_per_query * 1000, 3),
    'save_s': round(save_time, 4),
    'load_and_search_s': round(load_search_time, 4),
    'disk_mb': round(disk_mb, 2),
})

print(f' {name:12s} | {n:6d} vecs | insert={insert_time:.3f}s | '
      f'search={search_per_query*1000:.2f}ms/q | save={save_time:.3f}'
      f'load+search={load_search_time:.3f}s | disk={disk_mb:.1f}MB')

cleanup_fn(path)

return results

def dir_size_mb(path):
    total = 0
    if os.path.isfile(path):
        return os.path.getsize(path) / (1024*1024)
    if not os.path.exists(path):
        return 0
    for dp, dn, fns in os.walk(path):
        for f in fns:
            total += os.path.getsize(os.path.join(dp, f))
    return total / (1024*1024)

def cleanup_path(path):
    if os.path.isfile(path):
        os.remove(path)
    elif os.path.isdir(path):
        shutil.rmtree(path, ignore_errors=True)
    # También limpiar archivos con sufijos
    for suffix in ['.faiss', '.ids.npy', '.usearch', '.meta.json', '.db']:
        p = path + suffix
        if os.path.exists(p):
            os.remove(p)

print('Framework de benchmark listo')

```

Framework de benchmark listo

### 3. Implementaciones por backend

```

In [3]: # — FAISS —
import faiss

```

```

def faiss_insert(vecs, meta, path):
    index = faiss.IndexFlatIP(DIM)
    index.add(vecs)
    return index

def faiss_search(index, queries, k):
    scores, ids = index.search(queries, k)
    return ids

def faiss_save(index, path):
    faiss.write_index(index, path + '.faiss')

def faiss_load_search(path, queries, k):
    index = faiss.read_index(path + '.faiss')
    scores, ids = index.search(queries, k)
    return ids

def faiss_size(path):
    return dir_size_mb(path + '.faiss')

print('✓ FAISS listo')

```

✓ FAISS listo

```

In [4]: # — sqlite-vec —————
import sqlite3
import sqlite_vec

def sqlvec_insert(vecs, meta, path):
    db = sqlite3.connect(path + '.db')
    db.enable_load_extension(True)
    sqlite_vec.load(db)
    db.execute(f'CREATE VIRTUAL TABLE IF NOT EXISTS vec_items USING vec0(embedding)')
    # Insert en batches
    batch_size = 500
    for i in range(0, len(vecs), batch_size):
        batch = [(j, vecs[j].tobytes()) for j in range(i, min(i + batch_size, len(vecs)))]
        db.executemany('INSERT INTO vec_items(rowid, embedding) VALUES (?, ?)', batch)
        db.commit()
    return db

def sqlvec_search(db, queries, k):
    results = []
    for q in queries:
        rows = db.execute(
            'SELECT rowid, distance FROM vec_items WHERE embedding MATCH ? C'
            (q.tobytes(), k)
        ).fetchall()
        results.append([r[0] for r in rows])
    return results

def sqlvec_save(db, path):
    db.close() # Ya está persistido en el .db

def sqlvec_load_search(path, queries, k):

```

```

db = sqlite3.connect(path + '.db')
db.enable_load_extension(True)
sqlite_vec.load(db)
q = queries[0]
rows = db.execute(
    'SELECT rowid, distance FROM vec_items WHERE embedding MATCH ? ORDER
    (q.tobytes(), k)
').fetchall()
db.close()
return [r[0] for r in rows]

def sqlvec_size(path):
    return dir_size_mb(path + '.db')

print('✓ sqlite-vec listo')

```

✓ sqlite-vec listo

```

In [5]: # — LanceDB —————
import lancedb
import pyarrow as pa

def lance_insert(vecs, meta, path):
    db = lancedb.connect(path)
    data = pa.table({
        'id': list(range(len(vecs))),
        'category': [meta[i]['category'] for i in range(len(vecs))],
        'vector': [v.tolist() for v in vecs],
    })
    tbl = db.create_table('vectors', data, mode='overwrite')
    return tbl

def lance_search(tbl, queries, k):
    results = []
    for q in queries:
        r = tbl.search(q.tolist()).limit(k).to_list()
        results.append([row['id'] for row in r])
    return results

def lance_save(tbl, path):
    pass # LanceDB persiste automáticamente

def lance_load_search(path, queries, k):
    db = lancedb.connect(path)
    tbl = db.open_table('vectors')
    q = queries[0]
    r = tbl.search(q.tolist()).limit(k).to_list()
    return [row['id'] for row in r]

def lance_size(path):
    return dir_size_mb(path)

print('✓ LanceDB listo')

```

✓ LanceDB listo

```
In [6]: # — ChromaDB —————
import chromadb

def chroma_insert(vecs, meta, path):
    client = chromadb.PersistentClient(path=path)
    col = client.get_or_create_collection('vectors', metadata={'hnsw:space':
# Chroma tiene límite de batch, insertar en chunks
batch_size = 5000
    for i in range(0, len(vecs), batch_size):
        end = min(i + batch_size, len(vecs))
        col.add(
            ids=[str(j) for j in range(i, end)],
            embeddings=[v.tolist() for v in vecs[i:end]],
            metadatas=[meta[j] for j in range(i, end)],
        )
    return (client, col)

def chroma_search(state, queries, k):
    _, col = state
    results = col.query(query_embeddings=[q.tolist() for q in queries], n_re
    return results['ids']

def chroma_save(state, path):
    pass # PersistentClient persiste automáticamente

def chroma_load_search(path, queries, k):
    client = chromadb.PersistentClient(path=path)
    col = client.get_collection('vectors')
    results = col.query(query_embeddings=[queries[0].tolist()], n_results=k)
    return results['ids']

def chroma_size(path):
    return dir_size_mb(path)

print('✓ ChromaDB listo')
```

✓ ChromaDB listo

```
In [9]: # — USearch —————
from usearch.index import Index

def usearch_insert(vecs, meta, path):
    index = Index(ndim=DIM, metric='ip', dtype='f32')
    keys = np.arange(len(vecs), dtype=np.uint64)
    index.add(keys, vecs)
    return index

def usearch_search(index, queries, k):
    results = index.search(queries, k)
    return results.keys

def usearch_save(index, path):
    index.save(path + '.usearch')

def usearch_load_search(path, queries, k):
    index = Index(ndim=DIM, metric='ip', dtype='f32')
```

```

    index.load(path + '.usearch')
    results = index.search(queries[:1], k)
    return results.keys

def usearch_size(path):
    return dir_size_mb(path + '.usearch')

print('✓ USearch listo')

```

✓ USearch listo

## 4. Ejecutar benchmarks

```

In [8]: all_results = []

backends = [
    ('FAISS',      faiss_insert,  faiss_search,  faiss_save,  faiss_load),
    ('sqlite-vec', sqlvec_insert, sqlvec_search, sqlvec_save, sqlvec_load),
    ('LanceDB',    lance_insert,  lance_search, lance_save,  lance_load),
    ('ChromaDB',   chroma_insert, chroma_search, chroma_save, chroma_load),
    ('USearch',    usearch_insert, usearch_search, usearch_save, usearch_load)
]

for name, *fns in backends:
    print(f'\n— {name} —')
    try:
        res = bench_backend(name, *fns)
        all_results.extend(res)
    except Exception as e:
        print(f' ERROR: {e}')

df = pd.DataFrame(all_results)
print('\nBenchmark completo')

```

— FAISS —

FAISS		1000 vecs		insert=0.003s		search=0.23ms/q		save=0.006s
load+search=0.003s		disk=1.5MB						
FAISS		10000 vecs		insert=0.020s		search=0.48ms/q		save=0.030s
load+search=0.034s		disk=14.6MB						
FAISS		50000 vecs		insert=0.240s		search=2.77ms/q		save=0.169s
load+search=0.211s		disk=73.2MB						

— sqlite-vec —

sqlite-vec		1000 vecs		insert=0.053s		search=0.35ms/q		save=0.001s
load+search=0.001s		disk=1.6MB						
sqlite-vec		10000 vecs		insert=0.293s		search=4.87ms/q		save=0.000s
load+search=0.006s		disk=15.3MB						
sqlite-vec		50000 vecs		insert=1.275s		search=19.50ms/q		save=0.000s
load+search=0.018s		disk=74.7MB						

— LanceDB —

[2026-04-02T15:27:41Z **WARN** lance::dataset::write::insert] No existing dataset at /home/lucas/fn\_registry/analysis/estudio\_embeddings/notebooks/data/vector\_bench/LanceDB\_1000/vectors.lance, it will be created

```
LanceDB      |    1000 vecs | insert=0.040s | search=3.32ms/q | save=0.000s  
| load+search=0.007s | disk=1.5MB
```

```
[2026-04-02T15:27:42Z WARN lance::dataset::write::insert] No existing dataset at /home/lucas/fn_registry/analysis/estudio_embeddings/notebooks/data/vector_bench/LanceDB_10000/vectors.lance, it will be created
```

```
LanceDB      |   10000 vecs | insert=0.297s | search=12.30ms/q | save=0.000s  
| load+search=0.011s | disk=14.7MB
```

```
[2026-04-02T15:27:47Z WARN lance::dataset::write::insert] No existing dataset at /home/lucas/fn_registry/analysis/estudio_embeddings/notebooks/data/vector_bench/LanceDB_50000/vectors.lance, it will be created
```

```
LanceDB      |   50000 vecs | insert=3.740s | search=28.88ms/q | save=0.000s  
| load+search=0.030s | disk=73.7MB
```

— ChromaDB —

```
ChromaDB     |    1000 vecs | insert=0.492s | search=0.30ms/q | save=0.000s  
| load+search=0.006s | disk=4.1MB
```

```
ChromaDB     |   10000 vecs | insert=2.778s | search=0.53ms/q | save=0.000s  
| load+search=0.006s | disk=29.6MB
```

```
ChromaDB     |   50000 vecs | insert=19.123s | search=1.23ms/q | save=0.000s  
| load+search=0.010s | disk=108.6MB
```

— USearch —

```
ERROR: usearch.index.Index() got multiple values for keyword argument 'ndim'
```

Benchmark completo

## 5. Tabla resumen y visualizaciones

```
In [11]: df_display = df.copy()  
df_display.columns = ['Backend', 'Vectors', 'Insert (s)', 'Search 100q (s)',  
                     'Save (s)', 'Load+Search (s)', 'Disk (MB)']  
df_display.style.background_gradient(subset=['Per query (ms)', 'Insert (s)',
```



Out[11]:

	Backend	Vectors	Insert (s)	Search 100q (s)	Per query (ms)	Save (s)	Load+Search (s)
0	FAISS	1000	0.002700	0.022600	0.226000	0.006000	0.002600
1	FAISS	10000	0.020000	0.048000	0.480000	0.030200	0.034000
2	FAISS	50000	0.240500	0.277200	2.772000	0.169200	0.210900
3	sqlite-vec	1000	0.053100	0.035500	0.355000	0.000600	0.001300
4	sqlite-vec	10000	0.292600	0.486800	4.868000	0.000300	0.006000
5	sqlite-vec	50000	1.274600	1.949800	19.498000	0.000300	0.018300
6	LanceDB	1000	0.040000	0.332400	3.324000	0.000000	0.007100
7	LanceDB	10000	0.296800	1.230500	12.305000	0.000000	0.010700
8	LanceDB	50000	3.740100	2.888400	28.884000	0.000000	0.030100
9	ChromaDB	1000	0.491600	0.029500	0.295000	0.000000	0.006000
10	ChromaDB	10000	2.778100	0.053500	0.535000	0.000000	0.005500
11	ChromaDB	50000	19.122500	0.122800	1.228000	0.000000	0.009800
12	USearch	1000	0.010800	0.001300	0.013000	0.001100	0.001500
13	USearch	10000	0.289400	0.002500	0.025000	0.013400	0.018800
14	USearch	50000	5.965700	0.013500	0.135000	0.111700	0.188300

```
In [12]: fig, axes = plt.subplots(2, 2, figsize=(16, 12))
colors = {'FAISS': '#e74c3c', 'sqlite-vec': '#3498db', 'LanceDB': '#2ecc71',

# 1: Search latency per query
ax = axes[0, 0]
for name in df['backend'].unique():
    sub = df[df['backend'] == name]
    ax.plot(sub['n_vectors'], sub['per_query_ms'], 'o-', color=colors.get(name),
            label=name, linewidth=2, markersize=8)
ax.set_xlabel('Vectores en índice')
ax.set_ylabel('Latencia por query (ms)')
ax.set_title('Velocidad de búsqueda (top-10)')
ax.legend()
ax.set_xscale('log')
ax.set_yscale('log')

# 2: Insert time
ax = axes[0, 1]
for name in df['backend'].unique():
    sub = df[df['backend'] == name]
    ax.plot(sub['n_vectors'], sub['insert_s'], 's-', color=colors.get(name),
            label=name, linewidth=2, markersize=8)
ax.set_xlabel('Vectores')
ax.set_ylabel('Tiempo de inserción (s)')
ax.set_title('Velocidad de inserción')
ax.legend()
```

```

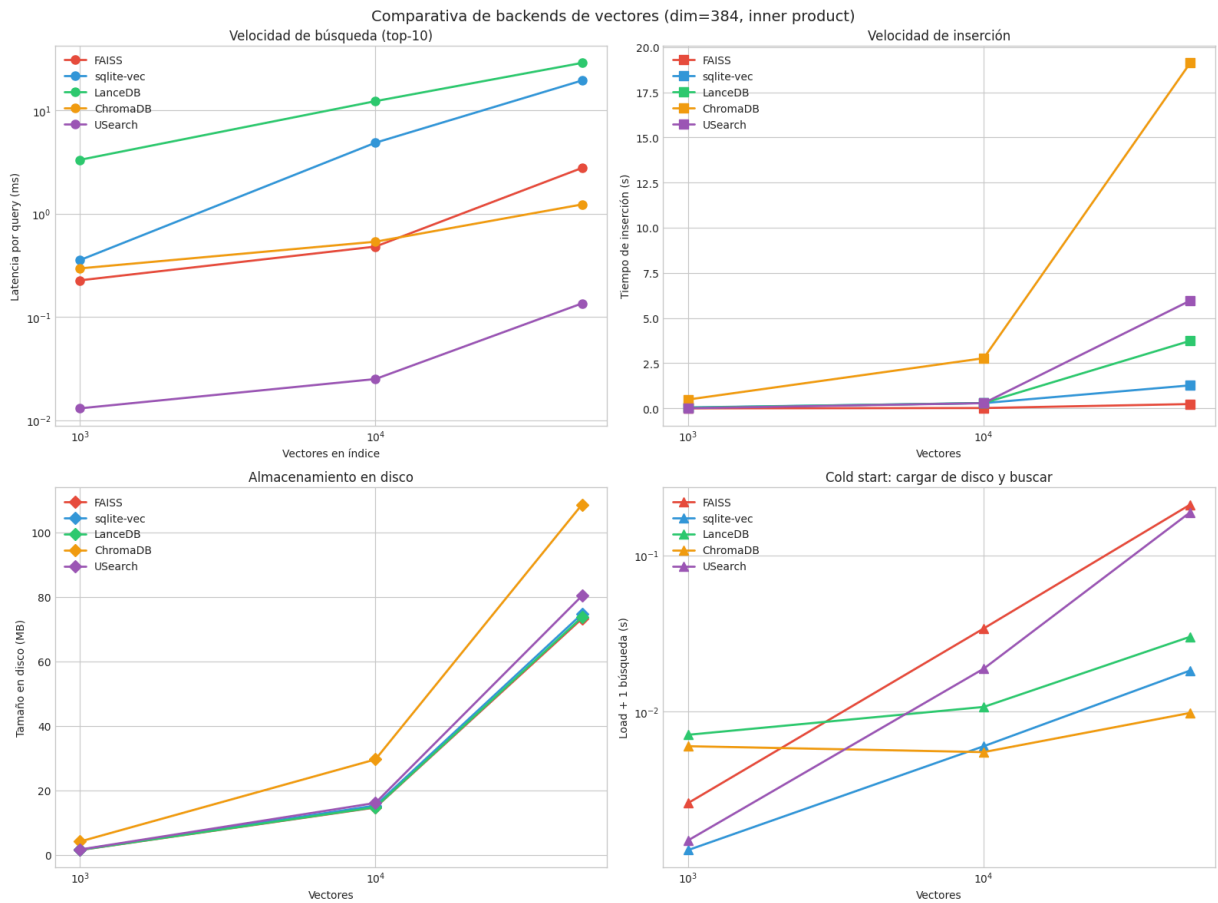
ax.set_xscale('log')

# 3: Disk usage
ax = axes[1, 0]
for name in df['backend'].unique():
    sub = df[df['backend'] == name]
    ax.plot(sub['n_vectors'], sub['disk_mb'], 'D-', color=colors.get(name, 'black'),
            label=name, linewidth=2, markersize=8)
ax.set_xlabel('Vectores')
ax.set_ylabel('Tamaño en disco (MB)')
ax.set_title('Almacenamiento en disco')
ax.legend()
ax.set_xscale('log')

# 4: Load from disk + search
ax = axes[1, 1]
for name in df['backend'].unique():
    sub = df[df['backend'] == name]
    ax.plot(sub['n_vectors'], sub['load_and_search_s'], '^-', color=colors.get(name, 'black'),
            label=name, linewidth=2, markersize=8)
ax.set_xlabel('Vectores')
ax.set_ylabel('Load + 1 búsqueda (s)')
ax.set_title('Cold start: cargar de disco y buscar')
ax.legend()
ax.set_xscale('log')
ax.set_yscale('log')

plt.suptitle('Comparativa de backends de vectores (dim=384, inner product)',
plt.tight_layout()
plt.show()

```



## 6. Resumen: pros y contras para fn\_registry

Backend	Pro	Contra	Ideal para
<b>FAISS</b>	Más rápido, battle-tested	Sin metadata nativa, hay que gestionar IDs aparte	Índices de solo vectores donde la velocidad importa
<b>sqlite-vec</b>	Mismo archivo SQLite, SQL completo, metadata con JOINS	Búsqueda más lenta en brute-force	Integrar vectores en registry.db o operations.db
<b>LanceDB</b>	Vector-native, filtros, columnar	Directorio no archivo único, API puede cambiar	Datasets medianos-grandes con filtros
<b>ChromaDB</b>	Metadata rica, popular	Más overhead, más lento, directorio grande	Prototipado rápido con metadata compleja
<b>USearch</b>	Ultra ligero, casi tan rápido como FAISS	Sin metadata, API minimalista	Índice puro de vectores, mínima dependencia

```
In [10]: # Re-run solo USearch
print('— USearch —')
try:
    usearch_res = bench_backend('USearch', usearch_insert, usearch_search, u
    all_results.extend(usearch_res)
```

```
df = pd.DataFrame(all_results)
print('USearch añadido al dataframe')
except Exception as e:
    import traceback
    traceback.print_exc()
```

— USearch —

USearch | 1000 vecs | insert=0.011s | search=0.01ms/q | save=0.001s  
| load+search=0.002s | disk=1.6MB

USearch | 10000 vecs | insert=0.289s | search=0.02ms/q | save=0.013s  
| load+search=0.019s | disk=16.1MB

USearch | 50000 vecs | insert=5.966s | search=0.13ms/q | save=0.112s  
| load+search=0.188s | disk=80.3MB

USearch añadido al dataframe