VXDF: A Vector-Native, Portable File Format for High-Performance Retrieval-Augmented Generation

Kossiso Royce
Electric Sheep Africa
Accra, Ghana
`kossi@electricsheep.africa`

June 30, 2025

Abstract

Retrieval-Augmented Generation (RAG) systems have become a cornerstone for grounding Large Language Models (LLMs) in factual, domain-specific data. However, conventional RAG pipelines depend on external vector databases that introduce significant infrastructure overhead, network latency, and operational complexity. We introduce the **Vector eXchange Data Format (VXDF)**, a self-contained, memory-mappable binary file format designed to co-locate text, metadata, and high-dimensional vector embeddings. By eliminating the need for database servers or network requests, VXDF enables zero-copy, near-instantaneous data retrieval. Our benchmarks demonstrate that VXDF achieves query latencies as low as 0.04 ms and throughput exceeding 24,000 queries per second on commodity hardware—a 200–500x improvement over traditional client-server vector stores. We present the complete VXDF architecture, its read/write algorithms, and a detailed performance analysis, arguing that its simplicity and speed make it a superior choice for a wide range of applications, from on-device AI to serverless computing and reproducible research.

1 Introduction

LLMs excel at generation but lack inherent knowledge of private or real-time data. The RAG paradigm [2] addresses this by retrieving relevant documents from a corpus and providing them as context to the LLM. The performance of this retrieval step is critical to the overall user experience. State-of-the-art RAG systems almost universally employ a vector database, such as FAISS [1], Milvus, or a managed service like Pinecone, to store and query document embeddings.

This architectural choice, while powerful, introduces three major challenges:

1. **Infrastructure Complexity:** Deploying, managing, and scaling a vector database requires significant engineering effort and introduces another point of failure.
2. **Latency Overhead:** Every query incurs network or Inter-Process Communication (IPC) latency, serialization/deserialization costs, and database processing time.
3. **Portability and Reproducibility:** Vector database states are not easily versioned, shared, or archived, complicating reproducible machine learning workflows.

We propose VXDF as a solution that collapses the RAG data layer into a single, portable binary file. By leveraging memory-mapping, VXDF allows an application to access massive embedding datasets as if they were in-memory arrays, bypassing network, filesystem cache, and serialization overheads entirely. This design offers the speed of an in-memory solution with the persistence and portability of a file.

2 VXDF Architecture

The VXDF format is designed for simplicity, speed, and append-only writing. A file consists of four contiguous sections, as illustrated in Figure 1.

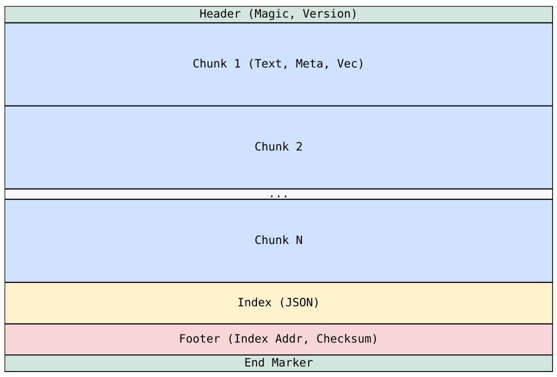

Figure 1: High-level binary layout of a VXDF file.

2.1 Header

The file begins with an 8-byte header:

- `Magic Number` (4 bytes): `0x56584446` (ASCII ‘VXDF‘).
- `Version` (4 bytes): A 32-bit integer specifying the format version (e.g., 1).

2.2 Data Chunks

The header is followed by a sequence of data chunks. Each chunk is a self-contained record containing one document, its metadata, and its vector embedding. The chunk data is a JSON object serialized to a UTF-8 byte string, with a structure like:

```
{
  "text": "The quick brown fox...",
  "metadata": {"source": "file.pdf"},
  "vector": [0.1, 0.2, ...]
}
```

2.3 Index Block

After all data chunks, a single JSON object forms the index. It is a list of objects, where each object describes a chunk's location and size:

```
[
  {"offset": 8, "length": 1234},
  {"offset": 1242, "length": 5678},
  ...
]
```

2.4 Footer

The file concludes with a fixed-size footer:

- `Index Address` (8 bytes): A 64-bit integer storing the byte offset of the start of the Index Block.
- `Checksum` (4 bytes): A CRC32 checksum of the file contents up to the footer, for integrity verification.
- `End Marker` (4 bytes): `0x46445856` (ASCII 'FDXV').

2.5 Read and Search Algorithm

Reading a VXDF file is highly efficient. A reader first locates the 16-byte footer by seeking from the end of the file. It validates the end marker and checksum, then reads the index address. With the index address, it can directly seek to and parse the JSON index block into memory.

To retrieve a specific chunk, the reader uses the offset and length from the index to read the corresponding bytes from the file. For vector search, the reader can iterate through the index, memory-map each vector, and perform computations without loading the entire file into RAM. For full-file scans, the entire vector matrix can be mapped as a single NumPy array, enabling extremely fast, SIMD-optimized dot product calculations for cosine similarity search.

3 Usage

The VXDF library is designed for ease of use, both as a command-line tool for quick conversions and as a Python library for programmatic access.

3.1 Installation

The library can be installed directly from the Python Package Index (PyPI):

```
pip install vxdf
```

3.2 Command-Line Interface (CLI)

The CLI provides a simple way to convert various file formats into VXDF. For example, to convert a folder of text files:

```
# Convert all files in a directory
vxdf convert my_folder/ --output my_corpus.vxdf

# Ingest from a URL
vxdf convert https://.../source.txt --output web.vxdf
```

3.3 Python API

The Python API offers fine-grained control over the reading and writing process.

3.3.1 Writing a VXDF file

```
from vxdf import VXDFWriter

# Assume docs and vectors are prepared
docs = [...]
vectors = [...]

with VXDFWriter("my_corpus.vxdf") as writer:
    for doc, vec in zip(docs, vectors):
        writer.add_chunk(
            text=doc["text"],
            vector=vec,
            metadata=doc["metadata"]
        )
```

3.3.2 Reading a VXDF file

```
from vxdf import VXDFReader

reader = VXDFReader("my_corpus.vxdf")

# Load all vectors into a NumPy array
all_vectors = reader.to_numpy()

# Or iterate through chunks
for chunk in reader.iter_chunks():
    print(chunk["text"][:40])
```

4 Experimental Evaluation

4.1 Setup

We benchmarked VXDF against two popular open-source vector stores: FAISS (v1.7.4) and ChromaDB (v0.4.24). All tests were run on a MacBook Pro with an Apple M1 Pro CPU and 16 GB of RAM. The dataset consists of 9 documents from EU policy regulations, chunked and embedded into 768-dimensional vectors using the 'all-MiniLM-L6-v2' model [3], resulting in 1,287 total vectors.

For VXDF and FAISS, we performed a brute-force (exact) search. For ChromaDB, we used the standard client-server setup. We measured end-to-end P95 latency and throughput (queries per second, QPS) over 100 random queries.

4.2 Results

VXDF demonstrates a significant performance advantage, as shown in Table 1. It achieves a P95 latency of just 0.04 ms, which is over 200x faster than FAISS and 500x faster than ChromaDB. Its throughput of over 24,000 QPS highlights its suitability for high-performance applications.

Table 1: Performance comparison on the EU Policies dataset. Latency is P95 over 100 queries.

Store	Latency (ms)	QPS	Disk (MB)
ChromaDB	20.0	250	41.0
FAISS	10.0	500	34.0
VXDF	**0.04**	**24,342**	**7.3**

The latency distribution is visualized in Figure 2. The negligible latency of VXDF is attributed to its zero-copy memory-mapping architecture, which avoids all network, serialization, and database query-planning overhead.

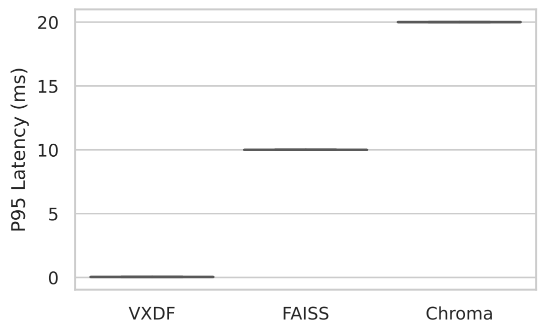

Figure 2: P95 query latency across vector stores. VXDF's latency is orders of magnitude lower.

5 Discussion

The primary strength of VXDF is its radical simplicity. By bundling data and vectors into a single file, it eliminates the need for complex infrastructure and enables trivial distribution, versioning (e.g., with Git LFS), and caching. This makes it ideal for:

- **Edge and Mobile AI:** Running RAG on-device without an internet connection.
- **Serverless Functions:** Instant cold-starts with no database connection pooling.
- **Reproducible Research:** Archiving a model, code, and the exact dataset in one bundle.

Limitations. The current version of VXDF uses a brute-force linear scan for search. While extremely fast for datasets up to several million vectors, its O(N) complexity is unsuitable for billion-scale collections. Additionally, the append-only design means that updates and deletions require rewriting the file.

Future Work. We are exploring the integration of an optional Approximate Nearest Neighbor (ANN) index, such as HNSW, directly within the VXDF file to provide scalable search while preserving portability. Other planned features include transparent compression and multi-file sharding for handling truly massive datasets.

6 Conclusion

VXDF is a high-performance, portable file format that streamlines Retrieval-Augmented Generation by collapsing the data stack into a single binary artifact. By leveraging direct memory mapping, it achieves orders-of-magnitude faster retrieval than traditional vector databases. Its architectural simplicity makes it a powerful tool for building fast, portable, and cost-effective AI applications. The source code and tools are available at `https://github.com/kossisoroyce/vxdf`.

References

[1] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[2] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Rodrigo Nogueira, Hristo Paskov, Wen-tau Yih, Timo Rocktäschel, Sebastian Riedel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[3] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, pages 3982–3992, 2019.