Vector Database Architecture Cheat Sheet
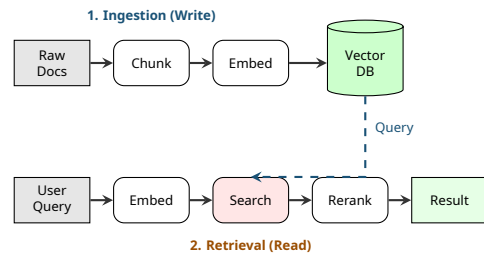
End-to-End Guide: From Data Ingestion to Query Serving

1. DEFINITION

Vector Database: A specialized database optimized for storing and querying high-dimensional vectors (embeddings).

- **Why Used:** Traditional DBs compare exact values (SQL 'WHERE id=5'). Vector DBs compare semantic similarity ('Find closest meaning') using distance metrics like Cosine Similarity. Essential for RAG and Semantic Search.

2. HIGH LEVEL ARCHITECTURE FLOW

1. Ingestion (Write)

Raw Docs → Chunk → Embed → Vector DB

Query

User Query → Embed → Search → Rerank → Result

2. Retrieval (Read)

- **Ingestion:** Raw data → Chunks → Vectors → Indexing.
- **Retrieval:** Query → Vector → ANN Search → Rerank → Response.

3. CORE COMPONENTS

- **Ingestion Layer:** Connectors (S3, SQL, PDF loaders) to fetch data.
- **Chunking/Preprocessing:** Splitting text into manageable windows (e.g., 512 tokens).
- **Embedding Layer:** Model (e.g., OpenAI text-embedding-3) converting text to floats.
- **Indexing Layer:** Organizes vectors for fast retrieval (IVF, HNSW).
- **Storage Layer:** Persists vectors, metadata, and the index structure.
- **Query Layer:** Handles search requests, filtering, and pre/post processing.
- **Reranking Layer:** Refines top-K results using a high-precision Cross-Encoder.

Chunking Strategy (LangChain):

```
from langchain.text_splitter import
    RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=500,        # Characters per chunk
    chunk_overlap=50,      # Context overlap
    separators=["\n\n", "\n", " ", ""]
)
chunks = text_splitter.split_text(raw_text)
```

4. EMBEDDING LAYER IMPLEMENTATION

Transformation

Raw Text "Apple" → Embedding Model → Encode → [0.12, -0.45, ..., 0.91]

High-Dimensional Vector
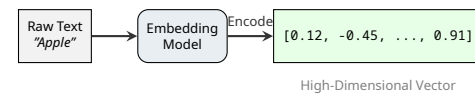
Providers

- **OpenAI:** Simple API, high quality, usage cost.
- **Sentence Transformers (HF):** Run locally (CPU/GPU), free, privacy-friendly.
- **Custom Fine-Tuned:** Best for domain-specific jargon (Medical/Legal).
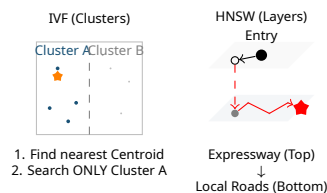
```
from sentence_transformers import SentenceTransformer

# Load a local model
model = SentenceTransformer('all-MiniLM-L6-v2')

# Create embeddings
text = "Vector databases allow semantic search."
vector = model.encode(text)
# Output: [0.034, -0.12, ... 384 dims]
```

5. INDEXING AND SEARCH ALGORITHMS

Algorithm Visuals

IVF (Clusters)

Cluster A Cluster B

1. Find nearest Centroid
2. Search ONLY Cluster A

HNSW (Layers)

Entry

Expressway (Top)
↓
Local Roads (Bottom)

Implementing FAISS (IndexFlatL2):

```
import faiss
import numpy as np

d = 384                              # Dimension
index = faiss.IndexFlatL2(d)         # Brute-force L2
index.add(embeddings_matrix)         # Add vectors

# Search
D, I = index.search(query_vector, k=5)
# D = Distances, I = Indices
```
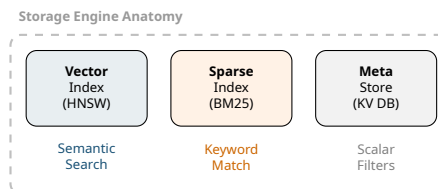
- **Flat Index:** Brute force. 100% Recall. Slow ($O(N)$).
- **IVF (Inverted File):** partitions space into clusters (Voronoi cells). Search only nearest clusters. Faster ($O(\sqrt{N})$), lower recall.
- **HNSW (Hierarchical Navigable Small World):** Multi-layer graph. Best trade-off for speed/recall. High memory usage.
- **PQ (Product Quantization):** Compresses vectors (lossy compression) to save RAM.
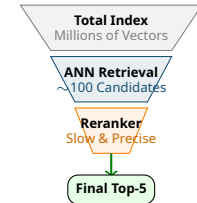
6. STORAGE ARCHITECTURE

Types

- **In-Memory (Redis/Milvus):** Ultra-low latency, expensive (RAM constrained).
- **Disk-Based (DiskANN):** Stores vectors on SSD. Slower, but scales to billions.
- **Metadata:** Stored separately (e.g., RocksDB, Postgres) for filtering.
- **Hybrid Search:** Storing both Dense Vectors and Sparse Vectors (Keywords/BM25) to handle exact matches.

Storage Engine Anatomy

Vector Index (HNSW)	Sparse Index (BM25)	Meta Store (KV DB)
Semantic Search	Keyword Match	Scalar Filters

7. QUERY PROCESSING

Query Pipeline (The Funnel)

Total Index — Millions of Vectors
ANN Retrieval — ~100 Candidates
Reranker — Slow & Precise
Final Top-5

Reranking (Cross-Encoder):

```
from sentence_transformers import CrossEncoder

reranker = CrossEncoder('cross-encoder/ms-marco-MiniLM-L-6-v2')
# (Query, Document) pairs
pairs = [('query', 'doc1_text'), ('query', 'doc2_text')]
scores = reranker.predict(pairs)

# Sort by score descending
ranked_results = sorted(zip(scores, docs), reverse=True)
```

- **1. Query Embed:** Convert user question to vector using same model as ingestion.
- **2. ANN Search:** Find approximate nearest neighbors (Top-K candidates).
- **3. Filtering:** Pre-filter (restrict search space) or Post-filter (remove results). Pre-filtering is faster but harder to implement with HNSW.
- **4. Reranking:** Re-score Top-K using a slower, precise model (Cross-Encoder).

8. VECTOR DATABASE IMPLEMENTATIONS

- **FAISS (Meta):** Library, not a DB. Good for local scripts. Fast.
- **Milvus/Zilliz:** Scalable, cloud-native DB. Good for massive datasets.
- **Weaviate:** Native hybrid search, modular (GraphQL).
- **Pinecone:** Fully managed SaaS. Easy to start, closed source.
- **Qdrant:** Rust-based, high performance, good filtering support.

- **Chroma:** Lightweight, developer-friendly, often local.

9. END TO END EXAMPLE FLOW

Scenario: User searches "Legal limitation for contracts."

1. **Query:** User input string.
2. **Embed:** OpenAI API converts string to 1536-dim vector.
3. **Search:** Vector DB scans HNSW index for nearest 100 neighbors.
4. **Filter:** Apply Metadata filter `doc_type == "legal"`.
5. **Rerank:** Cohere Rerank API re-sorts the 20 results by semantic relevance.
6. **Return:** Top 5 chunks returned to LLM context.

10. PERFORMANCE METRICS

- **Recall@K:** % of true nearest neighbors found in the top K results. (Quality).
- **Precision@K:** How many retrieved items are actually relevant.
- **Latency (p99):** Time taken for search. (Speed).
- **Throughput:** QPS (Queries Per Second). (Scale).
- **Index Build Time:** Time to index new data.
- **Memory Footprint:** RAM usage per million vectors.

Recall Calculation:

```python
def recall_at_k(actual_ids, predicted_ids, k):
    # Set intersection
    relevant_retrieved = set(actual_ids) & set(
        predicted_ids[:k])
    return len(relevant_retrieved) / len(actual_ids)
```
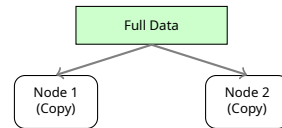
11. SCALING & OPTIMIZATION

Sharding vs Replication

1. Sharding (Partition)

2. Replication (Clone)

- **Sharding:** Split vectors across nodes to fit in memory (Horizontal Scaling).
- **Replication:** Copy index to multiple nodes to increase QPS (Throughput).

- **Async Updates:** Updates are eventually consistent to avoid blocking reads.

12. RELIABILITY & CONSISTENCY

- **Replication:** Ensures high availability if a node fails.
- **Backups:** Snapshotting the index to S3/Blob storage.
- **Consistency:** Most Vector DBs are *Eventually Consistent* (newly added vectors might not appear in search immediately) to favor speed.

13. COST OPTIMIZATION

Tiered Storage Hierarchy
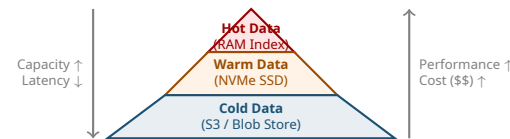
- **Tiered Storage:** Keep hot data (recent) in RAM, cold data (old) on Disk/SSD.
- **Index Compression:** Use PQ (Product Quantization) or Binary Quantization to reduce RAM usage by 10x-30x.
- **Separation:** Separate Compute nodes (search) from Storage nodes.

14. SECURITY & GOVERNANCE

- **RBAC:** Role-Based Access Control to restrict who can query collections.
- **Data Isolation:** Multi-tenancy (Metadata filtering vs Separate Indexes) for different users.
- **PII:** Redacting sensitive info before embedding (embeddings can theoretically be inverted).

15. COMMON FAILURE POINTS

- **Index Corruption:** Index file gets corrupted during write.
- **Embedding Drift:** Query model version doesn't match Ingestion model version (Silent failure).
- **Recall Degradation:** As index size grows or deletes happen, ANN accuracy drops (requires re-indexing).
- **Metadata Mismatch:** Filtering removes all relevant vector results.

16. SUMMARY

Goal: Efficiently store and retrieve knowledge based on meaning.
- **Architecture:** Embed → Index → Query.
- **Tradeoff:** Recall vs Latency vs Cost.
- **Key Tech:** HNSW (Graph Index), Quantization (Compression), Hybrid Search (Keyword + Vector).