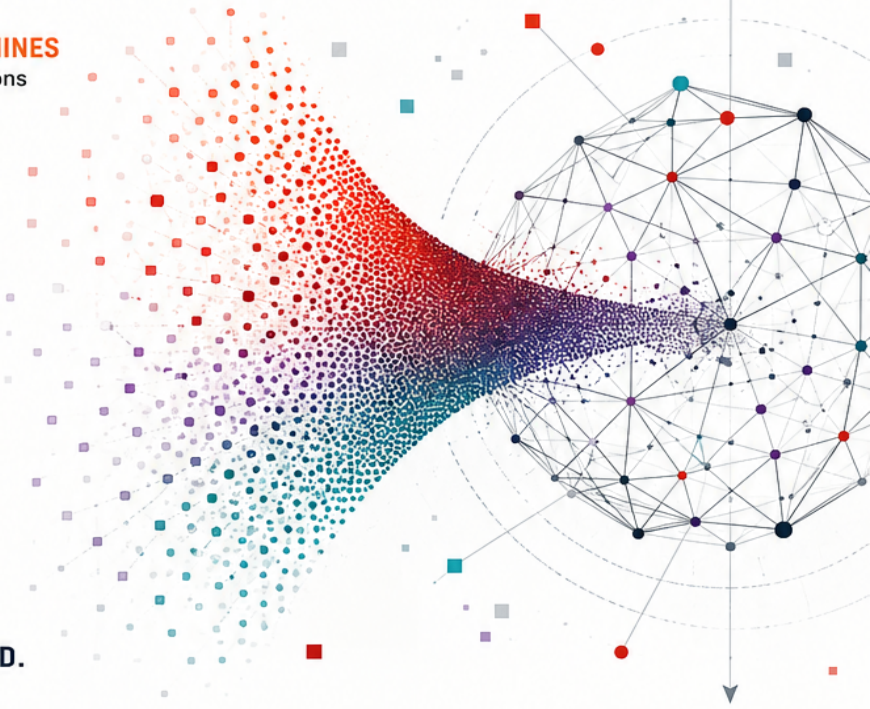
VECTOR SEARCH FROM FIRST PRINCIPLES

SIMD, QUANTIZATION, AND BILLION-SCALE RETRIEVAL



FROM MATH TO MACHINES
Understand the foundations
build with confidence

SIMD SPEED AT THE CORE
Vectorized kernels
that scream

QUANTIZE TO SCALE
Product quantization
that compresses
without compromise

HNSW FOR THE WIN
Approximate nearest
neighbor done right

DISTRIBUTED. DURABLE. DEPLOYED.
From single node to
billion-vector clusters

BUILD HIGH-PERFORMANCE RETRIEVAL SYSTEMS YOU UNDERSTAND. YOU OPTIMIZE. YOU SCALE.

MARCUS J. ZHAO

Vector Search from First Principles

SIMD, Quantization, and Billion-Scale Retrieval

Steve Publications

This book is available at
https://leanpub.com/vectorsearchfromfirstprinciples

This version was published on 2026-08-19

Contents

SIMD, Quantization, and Billion-Scale Retrieval **1**

Introduction: Why Vector Search Matters Now **2**
The Embedding Explosion 2
From Exact to Approximate: The Scaling Problem 3
What This Book Will Build 4
How to Read This Book 4

Chapter 1: Vectors, Embeddings, and Similarity **6**
What Is a Vector (in This Context)? 6
How Embeddings Are Created 7
Distance Metrics: Euclidean, Cosine, Dot Product 8
Similarity as a Geometric Problem 11
Starting the Running Implementation: Vector Data Structures 12

Chapter 2: Brute-Force k-NN and Its Limits **16**
The Exact Nearest-Neighbor Problem 16
Computational Complexity Analysis 16
Memory Bandwidth and the Real Bottleneck 16
Latency Budgets in Production Systems 16
Baseline Benchmarking Methodology 16

Chapter 3: Memory Hierarchy and Data Layout **18**
The Memory Wall: Registers, Caches, RAM, Disk 18
Cache Lines and Spatial Locality 18
Array of Structures vs Structure of Arrays for Vectors 18
Memory Alignment and Padding 18
NUMA Architecture and Its Impact 18

Chapter 4: Floating-Point Representation and Quantization Foundations **19**
IEEE 754 Floating-Point Format 19

The Cost of Float32 at Scale 19
Scalar Quantization (SQ) 19
Product Quantization (PQ): Subspace Compression 19
Binary Quantization and Hamming Distance 19

Chapter 5: Clustering-Based Indexes and IVF 20
The Partitioning Idea: Divide and Conquer in Vector Space 20
K-Means Clustering for Vector Indexes 20
Building an Inverted File Index (IVF) 20
IVF Search Parameters and Trade-offs 20
Extending the Running Implementation: IVF Index 20

Chapter 6: Graph-Based Search and HNSW 21
From Flat Graphs to Navigable Small Worlds 21
The HNSW Algorithm Explained 21
HNSW Parameters and Their Meaning 21
Memory Cost of HNSW Indexes 21
Extending the Running Implementation: HNSW Index 21

Chapter 7: Combining Techniques: Hybrid Approaches and Reranking . 22
IVF-PQ: The Industrial Standard Combination 22
HNSW with Quantization 22
Two-Tier Retrieval: Fast Recall, Accurate Reranking 22
Hybrid Search: Combining Vectors with Keywords and Metadata . . . 22
Extending the Running Implementation: Full Pipeline 22

Chapter 8: SIMD Vectorization and CPU-Level Optimization 23
What SIMD Actually Is 23
AVX2 Fundamentals: 256-Bit Registers and Instructions 23
Vectorizing Distance Computations 23
AVX-512: Doubling the Width, Adding Complexity 23
Benchmarking SIMD Speedups 23

Chapter 9: Parallelism, Concurrency, and Throughput 24
Multi-Core Threading for Vector Search 24
Lock-Free Index Reads and Safe Updates 24
Query Batching for Higher Throughput 24
Memory Mapping for Large Datasets 24
Extending the Running Implementation: Concurrent Server 24

Chapter 10: Persistence, Compression, and Disk-Backed Indexes **25**
Serializing Index Structures to Disk 25
Fast Loading and Warm-Start Strategies 25
Incremental Updates and Write-Ahead Logging 25
Compression Beyond Quantization 25
Extending the Running Implementation: Persistent Indexes 25

Chapter 11: Distributed Architecture and Sharding **26**
When Single-Machine Search Is No Longer Enough 26
Sharding Strategies for Vector Collections 26
Distributed Query Execution 26
Cross-Shard Recall and Its Degradation 26
Extending the Running Implementation: Multi-Node Cluster 26

Chapter 12: Observability, Benchmarking, and Capacity Planning **27**
Proper Benchmark Methodology for Vector Search 27
Monitoring Vector Search Systems in Production 27
Capacity Planning: From Thousands to Billions 27
Profiling and Debugging Performance Problems 27
Comparative Analysis of Production Vector Search Systems 27

Chapter 13: Vector Search in Modern AI Infrastructure **28**
Retrieval-Augmented Generation (RAG) Pipelines 28
Recommendation Systems and Vector Search 28
Multimodal Retrieval: Images, Audio, Text Together 28
The Future of Vector Search: Learned Indexes and Beyond 28

Conclusion: Principles Over Tools **29**
The Stack Revisited: From Bits to Clusters 29
Choosing the Right Tool for Your Problem 29
Building Systems You Understand 29
Final Words on the Running Implementation 29

References **30**

SIMD, Quantization, and Billion-Scale Retrieval

From floating-point arithmetic to billion-vector search clusters: building high-performance retrieval systems you understand. This book takes you from the fundamentals of vectors and similarity metrics through low-level CPU optimization, indexing algorithms, and distributed systems engineering. Along the way we build a running vector-search implementation that evolves from naive brute-force search into a production-ready approximate nearest-neighbor engine with HNSW indexing, product quantization, SIMD acceleration, persistence, and multi-node scaling. No prior expertise in vector databases is required: if you can write code and understand basic linear algebra, you will finish this book able to implement, profile, optimize, and deploy vector retrieval at scale.

Introduction: Why Vector Search Matters Now

The Embedding Explosion

A few years ago, searching for information meant matching keywords. You typed words into a search box, the system found documents containing those same words, and ranked them by frequency and position. It was brittle but understandable. Today, that model has largely been replaced by something stranger and more powerful: you ask a question in natural language, and the system returns answers based on meaning rather than word overlap.

The engine behind this shift is not magic. It is vector search.

Modern AI models called embedding models take text, images, audio, or any structured data and convert it into dense arrays of numbers called vectors. A typical text embedding might be a list of 768 floating-point values that collectively encode the semantic content of a paragraph. Two paragraphs about the same topic will produce similar vectors regardless of whether they share vocabulary. This geometric representation of meaning transforms search from pattern matching into geometry: finding relevant information becomes a problem of locating nearby points in high-dimensional space.

The applications have proliferated rapidly. Recommendation engines use vector similarity to find products or content similar to what users already like. Retrieval-augmented generation pipelines retrieve relevant documents before feeding them to large language models, grounding generated answers in actual data rather than training memory. Multimodal search lets you find images with text queries or vice versa, because CLIP-style models embed both modalities into a shared vector space where semantic proximity is preserved. Vector search has become critical infrastructure for AI applications the way relational databases became critical infrastructure for business software in the 1980s.

But here is what most practitioners do not realize: choosing a vector database and calling its API is not enough to build robust, scalable systems. When your collection grows from thousands of vectors to millions or billions,

when your latency budget shrinks from hundreds of milliseconds to single digits, when you need to combine semantic search with metadata filtering and keyword matching, the surface-level abstractions start to leak. You need to understand what is happening underneath.

From Exact to Approximate: The Scaling Problem

The simplest possible vector search algorithm is embarrassingly straightforward. Given a query vector and a collection of stored vectors, compute the distance between the query and every stored vector, then return the k closest ones. This is exact k-nearest neighbors (k-NN) search via brute-force exhaustive scan. For small collections it works perfectly and guarantees correct results.

The problem is scale. Each distance computation for two 768-dimensional vectors requires roughly 768 floating-point multiplications, 768 subtractions, and a summation. Search through one million vectors and you need about 768 million floating-point operations per query. Search through one billion vectors and that becomes 768 billion operations. Even on fast hardware, doing this for every query in real time is unsustainable.

But the computational cost is actually not the worst part. The real bottleneck is memory bandwidth. To compute distances, you must stream vector data from RAM into CPU caches. Modern servers might have 200 to 600 gigabytes per second of memory bandwidth. One billion vectors at 768 dimensions in float32 format occupies roughly three terabytes. Streaming through all that data for a single query would take several seconds even if the CPU could compute distances instantly. The math is unforgiving: exact search does not scale to billion-vector collections on practical hardware.

The solution is approximation. Instead of comparing against every vector, we use indexing structures that let us skip most of them while still finding results that are close enough. These approximate nearest neighbor (ANN) algorithms trade a small amount of accuracy for orders-of-magnitude speedup. At scale this is not a compromise but an engineering necessity: you cannot serve real-time queries over billion-vector collections without approximation, period.

The question becomes which approximation technique to use and how to tune it. Different algorithms make different trade-offs between recall (how

many of the truly nearest neighbors do we actually find), latency (how fast does each query return), memory usage (how much RAM does the index consume), and build time (how long does it take to construct the index). Understanding these trade-offs requires understanding both the algorithms themselves and the hardware they run on.

What This Book Will Build

This book is organized around a running implementation: a vector-search engine written in C++ that we develop incrementally from chapter to chapter. We start with the simplest possible exact search implementation and progressively add capabilities until it becomes a production-ready ANN system. Each chapter contributes specific features:

Early chapters establish foundations. We define vectors, similarity metrics, and distance functions with mathematical precision. We implement brute-force k-NN search and analyze its complexity rigorously. We examine the hardware reality of CPU caches, memory hierarchy, and floating-point representation that every optimization must contend with.

Middle chapters introduce indexing algorithms. We build a clustering-based inverted file index (IVF) using k-means clustering to partition vector space. We implement hierarchical navigable small world graphs (HNSW), arguably the most important ANN algorithm in current production use. We add product quantization to compress vectors by factors of 8x to 96x while preserving search quality.

Later chapters focus on systems engineering. We accelerate distance computation using SIMD vector instructions (AVX2 and AVX-512) that let a single CPU instruction operate on multiple data elements simultaneously. We add multi-threading, query batching, and NUMA-aware memory placement to exploit modern multi-core servers. We implement persistence so indexes survive restarts, compression to reduce storage costs, and distributed scaling across multiple machines for billion-vector clusters.

By the end you will have a working vector-search system with indexing, quantization, SIMD acceleration, concurrency, persistence, filtering support, and distributed capability. More importantly you will understand why each design choice was made, what alternatives exist, and how to reason about trade-offs when building your own systems or evaluating existing tools.

How to Read This Book

This book assumes general programming competence: comfort with at least one compiled language (C++ is used for the implementation), basic understanding of data structures and algorithms, and familiarity with linear algebra concepts like vectors and dot products. No prior experience with vector databases, ANN algorithms, or SIMD programming is required. Every concept is introduced from first principles.

The chapters build on each other in a deliberate sequence. Chapter 1 defines the mathematical objects we will manipulate. Chapter 2 establishes performance baselines through brute-force search. Chapters 3 and 4 explain the hardware constraints that motivate optimization. Chapters 5 through 7 introduce indexing and compression algorithms. Chapters 8 through 10 focus on low-level systems optimization. Chapters 11 through 13 cover distributed architecture and production operations.

You can read the book linearly from beginning to end for a comprehensive education in vector search internals. If you already understand some topics you can skim early chapters and dive into later material. The running implementation code is available alongside each chapter: read the explanation first, then study the code, then experiment by modifying it yourself.

One note on style: this book prioritizes deep understanding over superficial coverage. We will not survey every vector database product or list every API option. Instead we focus on timeless principles: how distance computation works at the bit level, how indexing structures organize vector space, how hardware constraints shape algorithm design. Products come and go but these fundamentals endure. When you understand them deeply, you can evaluate any tool critically and build systems that genuinely fit your requirements rather than accepting defaults blindly.

Let us begin.

Chapter 1: Vectors, Embeddings, and Similarity

What Is a Vector (in This Context)?

In the context of vector search, a vector is a fixed-length array of numbers that represents some piece of data in a geometric space. Each number in the array is called a dimension or coordinate. When we talk about searching vectors, we are really talking about finding points in this geometric space that are close to a query point according to some notion of distance.

Consider a simple example. Imagine representing fruits as two-dimensional vectors where the first coordinate encodes sweetness and the second encodes acidity:

- Apple: (0.7, 0.4)
- Lemon: (0.1, 0.9)
- Orange: (0.6, 0.5)
- Grapefruit: (0.3, 0.8)

In this two-dimensional space, apples and oranges are geometrically close because they have similar sweetness and acidity profiles. Lemons and grapefruits cluster together as sour fruits. This is the fundamental insight behind vector search: if you can represent data as points in a space where geometric proximity corresponds to semantic similarity, then finding similar items becomes a geometry problem.

Real-world vectors are not two-dimensional. Typical text embeddings from models like OpenAI's text-embedding-3-small or sentence-transformers have 768 to 1536 dimensions. Multimodal models like CLIP produce 768-dimensional vectors for both images and text in a shared space. Some specialized models go higher: certain bioinformatics embeddings exceed 4000 dimensions. Despite the dimensionality, the geometric intuition remains the same: similar things cluster together, dissimilar things are far apart.

Each dimension in a high-dimensional embedding does not usually correspond to an easily interpretable feature like "sweetness." Instead, the dimensions emerge from the training process as abstract directions in a space that collectively encode complex patterns. You cannot meaningfully read off what the 347th dimension represents, but you can trust that the overall geometry preserves semantic relationships. This is both powerful and somewhat mysterious: the model has discovered a representation where cosine similarity between vectors correlates with human judgment of textual similarity, even though no individual dimension maps to a named concept.

For our purposes, a vector is simply an array of d floating-point numbers. We will use float32 (32-bit IEEE 754 single-precision) as our standard format because it offers a good balance between numerical precision and memory efficiency. A single 768-dimensional float32 vector occupies 768 * 4 = 3072 bytes, roughly three kilobytes. One million such vectors occupy about three gigabytes. This concrete accounting of bytes matters enormously when we discuss scaling later in the book.

How Embeddings Are Created

Embeddings are produced by neural network models called encoders. The input can be text, images, audio waveforms, tabular data, or any structured format. The output is always a fixed-length vector regardless of input size: a single word and an entire document both produce vectors with the same number of dimensions. This normalization is essential for search because it lets us compare items of vastly different sizes using the same distance metric.

The most common text embedding models today fall into several families. Word-level models like word2vec and GloVe (introduced around 2013 to 2014) produce one vector per vocabulary word, capturing distributional similarity: words that appear in similar contexts get similar vectors. Sentence-level models based on transformer architectures, such as Sentence-BERT and its successors, encode entire sentences or paragraphs into single vectors while preserving semantic meaning. OpenAI's embedding models (text-embedding-ada-002, text-embedding-3-small, text-embedding-3-large) are highly optimized proprietary encoders widely used in production systems.

Multimodal models like CLIP train on image-text pairs to learn a shared embedding space where semantically matching images and captions are close

together. This enables cross-modal search: you can query with text and retrieve relevant images, or query with an image and find similar images or descriptive text, all using the same underlying vector similarity computation.

For this book we do not need to understand how these models are trained in detail. What matters is their output format and properties:

1. **Fixed dimensionality**: Every embedding from a given model has exactly d dimensions.
2. **Density**: Unlike sparse representations (where most values are zero), embeddings are dense arrays where nearly every dimension carries information.
3. **Normalization**: Many models produce vectors that are normalized to unit length, meaning the sum of squared components equals one. This is important for similarity metrics as we will see shortly.
4. **Direction over magnitude**: For normalized vectors, semantic information is encoded in the direction of the vector rather than its length. Two texts about the same topic will point in similar directions regardless of document length or emphasis.

These properties are not accidental: they emerge from training objectives that explicitly reward geometric arrangements where similarity corresponds to proximity. Contrastive learning losses push embeddings of related items closer together while pushing unrelated items apart. The resulting geometry is what vector search exploits.

Distance Metrics: Euclidean, Cosine, Dot Product

To find the nearest neighbors of a query vector, we need a way to measure how far apart two vectors are. Different distance metrics capture different notions of similarity, and choosing the right one matters for search quality. The three most common metrics in practice are Euclidean distance, cosine similarity, and dot product. Each has distinct mathematical properties and appropriate use cases.

Euclidean Distance

Euclidean distance is the straight-line distance between two points in space. For two d-dimensional vectors x and y, it is defined as:

d_Euclidean(x, y) = sqrt(sum of (x_i - y_i)^2 for i from 1 to d)

This is the familiar Pythagorean distance generalized to arbitrary dimensions. In our fruit example above, the Euclidean distance between apple (0.7, 0.4) and orange (0.6, 0.5) is:

sqrt((0.7 - 0.6)$^{2 + (0.4 - 0.5)}$2) = sqrt(0.01 + 0.01) = sqrt(0.02) ≈ 0.141

The distance between apple and lemon is:

sqrt((0.7 - 0.1)$^{2 + (0.4 - 0.9)}$2) = sqrt(0.36 + 0.25) = sqrt(0.61) ≈ 0.781

Apples are closer to oranges than to lemons, which matches our intuition. For Euclidean distance, smaller values indicate greater similarity: the nearest neighbor has the smallest distance.

Euclidean distance treats both direction and magnitude as meaningful. If one vector is twice as long as another in every dimension, they will have a large Euclidean distance even though they point in the same direction. This makes Euclidean distance appropriate when absolute scale carries semantic information.

Cosine Similarity

Cosine similarity measures the cosine of the angle between two vectors, ignoring their magnitudes. It is defined as:

cosine(x, y) = (x dot y) / (||x|| * ||y||)

where x dot y is the dot product (sum of x_i * y_i) and ||x|| is the Euclidean norm (sqrt of sum of x_i^2).

The cosine of an angle ranges from -1 to +1. A value of +1 means the vectors point in exactly the same direction (angle of zero degrees), -1 means they point in opposite directions, and 0 means they are orthogonal (perpendicular). For text embeddings where all values tend to be positive, cosine similarity typically falls between 0 and 1.

Using our fruit example:

- Apple dot orange = 0.70.6 + 0.40.5 = 0.42 + 0.20 = 0.62
- ||apple|| = sqrt(0.7$^{2 + 0.4}$2) = sqrt(0.65) ≈ 0.806
- ||orange|| = sqrt(0.6$^{2 + 0.5}$2) = sqrt(0.61) ≈ 0.781
- cosine(apple, orange) = 0.62 / (0.806 * 0.781) ≈ 0.984

Cosine similarity is the dominant metric for text-based semantic search because it isolates direction from magnitude. Two documents about the same topic will have similar directional profiles even if one is much longer than the other. Most modern sentence embedding models are trained with cosine similarity as the target metric, meaning their geometry is optimized for angular rather than Euclidean proximity.

For cosine similarity, larger values indicate greater similarity: the nearest neighbor has the highest cosine score. Many systems convert this to a distance by computing 1 - cosine(x, y), so that smaller distances again mean more similar items.

Dot Product Similarity

The dot product (also called inner product) is the numerator of the cosine formula without normalization:

dot(x, y) = sum of (x_i * y_i for i from 1 to d)

For unnormalized vectors, the dot product captures both directional alignment and magnitude. Two long vectors pointing in similar directions will have a large dot product even if their angle is moderate. This can be desirable when magnitude encodes meaningful information: for example, in recommendation systems where vector length might represent user activity level or item popularity.

Here is the crucial relationship: if both vectors are normalized to unit length (||x|| = ||y|| = 1), then the dot product equals cosine similarity exactly:

dot(x, y) = cosine(x, y) when ||x|| = ||y|| = 1

Many embedding models produce normalized outputs precisely to enable this simplification. When vectors are unit-length, you can use the faster dot product computation instead of the more expensive cosine formula (which requires computing norms). This is a common optimization in production systems: normalize all stored vectors once during indexing, then use dot product for all queries.

When to Use Which Metric

The choice depends on your embedding model and what properties matter for your application:

- **Cosine similarity**: Default choice for text embeddings from most modern models (Sentence-BERT, E5, GTE, OpenAI embeddings). Use when you care about semantic direction rather than document length or intensity.
- **Dot product**: Use when your vectors are already normalized (equivalent to cosine but faster), or when magnitude carries meaning in your domain. Many vector databases optimize dot product search specifically because of its computational simplicity.
- **Euclidean distance**: Use when absolute differences in feature values matter, such as comparing physical measurements, sensor readings, or any domain where scale is semantically meaningful. Also common for image features like SIFT descriptors that were not trained with angular objectives.

A practical rule: match your distance metric to what your embedding model was trained with. If a model uses contrastive loss based on cosine similarity during training, use cosine similarity at search time. Mismatching the metric can degrade retrieval quality even if the embeddings themselves are high-quality.

Similarity as a Geometric Problem

Visualizing vectors in two or three dimensions helps build intuition for what happens in hundreds or thousands of dimensions. In low-dimensional space, nearest neighbor search is geometrically straightforward: you have points scattered around, and finding the closest one to a query point means looking at its immediate neighborhood.

As dimensionality increases, several counterintuitive phenomena emerge that affect search algorithms. The most famous is called the curse of dimensionality. In very high-dimensional spaces, distances between random points tend to concentrate: the ratio between the distance to the nearest neighbor and the distance to the farthest neighbor approaches one. Intuitively, this means all points start looking roughly equally distant from each other, making discrimination difficult.

However, real-world embeddings do not behave like random points. They lie on or near low-dimensional manifolds within the high-dimensional space: the effective number of independent directions carrying information is much smaller than the raw dimensionality. A 768-dimensional text embedding might

have an intrinsic dimensionality of only 50 to 100 meaningful directions, with the remaining dimensions encoding redundancy or noise. This structure is what makes ANN algorithms work in practice despite the theoretical curse: the data is not uniformly distributed across all dimensions but clustered along meaningful semantic axes.

The curse of dimensionality does have practical consequences though. Exact search methods that partition space using trees (like k-d trees) degrade to nearly linear scan performance in high dimensions because the partitions become ineffective at pruning irrelevant regions. This is why tree-based methods are rarely used for modern embedding search: clustering-based indexes (IVF) and graph-based indexes (HNSW) handle high-dimensional structure more robustly.

Starting the Running Implementation: Vector Data Structures

We now begin our running implementation: a vector-search library written in C++ that we will extend throughout this book. Our first task is defining basic data structures for storing vectors and computing distances. This code establishes conventions we will use going forward.

We represent a collection of vectors as a contiguous array of float values in row-major order: all dimensions of vector 0, followed by all dimensions of vector 1, and so on. This layout maximizes cache efficiency when scanning through vectors sequentially, which is exactly what brute-force search does.

```cpp
// vecsearch.h - Basic vector storage and distance computation

#include <vector>
#include <cmath>
#include <algorithm>
#include <utility>
#include <cstdint>

namespace vecsearch {

// A simple flat index storing vectors in a contiguous array
class FlatIndex {
public:
    // dimension: number of dimensions per vector (d)
```

```
    FlatIndex(int dimension) : dim_(dimension), data_() {}

    // Add a single vector. Returns its assigned ID.
    int add(const float* vec) {
        int id = static_cast<int>(data_.size()) / dim_;
        data_.insert(data_.end(), vec, vec + dim_);
        return id;
    }

    // Add multiple vectors at once from a contiguous array
    void add_many(int count, const float* data) {
        data_.insert(data_.end(), data, data + count * dim_);
    }

    // Return number of stored vectors
    int size() const {
        return static_cast<int>(data_.size()) / dim_;
    }

    // Return dimensionality
    int dim() const { return dim_; }

    // Compute squared L2 distance between two vectors at given IDs
    float l2_distance(int id_a, int id_b) const {
        float dist = 0.0f;
        for (int i = 0; i < dim_; ++i) {
            float diff = data_[id_a * dim_ + i] - data_[id_b * dim_ + i];
            dist += diff * diff;
        }
        return dist;
    }

    // Compute dot product between two vectors at given IDs
    float dot_product(int id_a, int id_b) const {
        float sum = 0.0f;
        for (int i = 0; i < dim_; ++i) {
            sum += data_[id_a * dim_ + i] * data_[id_b * dim_ + i];
        }
        return sum;
    }

    // Brute-force k-nearest neighbors using L2 distance
    // Returns pairs of (distance, vector_id) for the k closest vectors
    std::vector<std::pair<float, int>> search_l2(
        const float* query, int k) const {

        if (k > size()) k = size();

        // Use a max-heap to track top-k smallest distances
        // Store negative distances because priority_queue is a max-heap
```

```
        std::priority_queue<std::pair<float, int>> heap;

        for (int i = 0; i < size(); ++i) {
            float dist = 0.0f;
            const float* vec = &data_[i * dim_];
            for (int d = 0; d < dim_; ++d) {
                float diff = query[d] - vec[d];
                dist += diff * diff;
            }

            if (heap.size() < static_cast<size_t>(k)) {
               heap.push({-dist, i});  // negative for max-heap -> min-distance
            } else if (-dist < heap.top().first) {
                heap.pop();
                heap.push({-dist, i});
            }
        }

        // Extract results in order (smallest distance first)
        std::vector<std::pair<float, int>> results;
        results.reserve(k);
        while (!heap.empty()) {
            results.push_back({-heap.top().first, heap.top().second});
            heap.pop();
        }
        std::reverse(results.begin(), results.end());
        return results;
    }

private:
    int dim_;
    std::vector<float> data_;  // contiguous: [v0_dim0, v0_dim1, ..., v1_dim0,
    ↪  ...]
};

}  // namespace vecsearch
```

This implementation is deliberately simple and correct rather than fast. The search_l2 function computes the squared L2 distance from the query to every stored vector using a straightforward nested loop. We use squared distance (without the square root) because it preserves ordering: if vector A has smaller squared distance than vector B, then A also has smaller actual distance. Skipping the square root saves expensive operations without affecting which vectors are returned as nearest neighbors.

The top-k selection uses a max-heap that maintains at most k entries. As we scan through all vectors, we keep the k smallest distances seen so far by discarding any new distance larger than the current worst in our heap. This

approach uses O(k) extra space regardless of collection size and avoids sorting all n distances (which would cost O(n log n)).

To test this implementation:

```cpp
#include <iostream>
#include <random>

int main() {
    const int DIM = 128;
    const int N_VECTORS = 10000;
    const int K = 5;

    vecsearch::FlatIndex index(DIM);

    // Generate random vectors for testing
    std::mt19937 rng(42);
    std::normal_distribution<float> dist(0.0f, 1.0f);
    std::vector<float> temp(DIM);

    for (int i = 0; i < N_VECTORS; ++i) {
        for (int d = 0; d < DIM; ++d) {
            temp[d] = dist(rng);
        }
        index.add(temp.data());
    }

    // Create a query vector near the first stored vector
    std::vector<float> query(DIM);
    for (int d = 0; d < DIM; ++d) {
        query[d] = temp[d] + dist(rng) * 0.1f;  // small perturbation
    }

    // Search
    auto results = index.search_l2(query.data(), K);

    std::cout << "Top " << K << " nearest neighbors:" << std::endl;
    for (const auto& [dist, id] : results) {
        std::cout << "  ID " << id << ", distance^2 = " << dist << std::endl;
    }

    return 0;
}
```

This baseline implementation is our starting point. It is correct but slow: every query examines all stored vectors with no shortcuts. In the next chapter we will measure its performance rigorously and discover exactly why it fails at scale, setting the stage for every optimization that follows.

Chapter 2: Brute-Force k-NN and Its Limits

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

The Exact Nearest-Neighbor Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Computational Complexity Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Memory Bandwidth and the Real Bottleneck

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Latency Budgets in Production Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Baseline Benchmarking Methodology

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Warm-Up

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Repeated Measurements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Controlled Conditions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Standard Metrics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Implementation Sketch

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Chapter 3: Memory Hierarchy and Data Layout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

The Memory Wall: Registers, Caches, RAM, Disk

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Cache Lines and Spatial Locality

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Array of Structures vs Structure of Arrays for Vectors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Memory Alignment and Padding

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

NUMA Architecture and Its Impact

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Chapter 4: Floating-Point Representation and Quantization Foundations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

IEEE 754 Floating-Point Format

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

The Cost of Float32 at Scale

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Scalar Quantization (SQ)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Product Quantization (PQ): Subspace Compression

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Binary Quantization and Hamming Distance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Chapter 5: Clustering-Based Indexes and IVF

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

The Partitioning Idea: Divide and Conquer in Vector Space

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

K-Means Clustering for Vector Indexes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Building an Inverted File Index (IVF)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

IVF Search Parameters and Trade-offs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Extending the Running Implementation: IVF Index

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Chapter 6: Graph-Based Search and HNSW

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

From Flat Graphs to Navigable Small Worlds

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

The HNSW Algorithm Explained

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

HNSW Parameters and Their Meaning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Memory Cost of HNSW Indexes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Extending the Running Implementation: HNSW Index

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Chapter 7: Combining Techniques: Hybrid Approaches and Reranking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

IVF-PQ: The Industrial Standard Combination

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

HNSW with Quantization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Two-Tier Retrieval: Fast Recall, Accurate Reranking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Hybrid Search: Combining Vectors with Keywords and Metadata

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Extending the Running Implementation: Full Pipeline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Chapter 8: SIMD Vectorization and CPU-Level Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

What SIMD Actually Is

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

AVX2 Fundamentals: 256-Bit Registers and Instructions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Vectorizing Distance Computations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

AVX-512: Doubling the Width, Adding Complexity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Benchmarking SIMD Speedups

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Chapter 9: Parallelism, Concurrency, and Throughput

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Multi-Core Threading for Vector Search

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Lock-Free Index Reads and Safe Updates

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Query Batching for Higher Throughput

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Memory Mapping for Large Datasets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Extending the Running Implementation: Concurrent Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Chapter 10: Persistence, Compression, and Disk-Backed Indexes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Serializing Index Structures to Disk

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Fast Loading and Warm-Start Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Incremental Updates and Write-Ahead Logging

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Compression Beyond Quantization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Extending the Running Implementation: Persistent Indexes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Chapter 11: Distributed Architecture and Sharding

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

When Single-Machine Search Is No Longer Enough

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Sharding Strategies for Vector Collections

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Distributed Query Execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Cross-Shard Recall and Its Degradation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Extending the Running Implementation: Multi-Node Cluster

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Chapter 12: Observability, Benchmarking, and Capacity Planning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Proper Benchmark Methodology for Vector Search

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Monitoring Vector Search Systems in Production

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Capacity Planning: From Thousands to Billions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Profiling and Debugging Performance Problems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Comparative Analysis of Production Vector Search Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Chapter 13: Vector Search in Modern AI Infrastructure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Retrieval-Augmented Generation (RAG) Pipelines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Recommendation Systems and Vector Search

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Multimodal Retrieval: Images, Audio, Text Together

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

The Future of Vector Search: Learned Indexes and Beyond

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Conclusion: Principles Over Tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

The Stack Revisited: From Bits to Clusters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Choosing the Right Tool for Your Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Building Systems You Understand

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

Final Words on the Running Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/vectorsearchfromfirstprinciples.