



Shahid Beheshti University (SBU)

Department of Computer Science

Data Structures and Algorithms

Content-Based Image Retrieval Using
Locality Sensitive Hashing

Instructor: Dr. Ali Katanforoosh

Fall 2025

Abstract

In this project, you are tasked with implementing a comprehensive Content-Based Image Retrieval (CBIR) system using Vector Database and Locality Sensitive Hashing (LSH) techniques. The project requires you to:

- Design and implement a vector storage system (basic Vector Database with CRUD operations)
- Use pre-trained deep learning library ResNet18 to generate embedding vectors for your input images
- Implement exact k-NN search using multiple distance metrics
- Research and understand the theoretical foundations of Locality Sensitive Hashing
- Implement LSH algorithm for approximate nearest neighbor search
- Develop a user-friendly GUI for image similarity search
- Compare and analyze the performance of exact versus approximate search methods

1. The Basics:

1.1 What is a “Vector DB”?

A Vector Database is a storage system optimized for numerical data representations. Embedding models convert text, images, or audio into numerical vectors, which the database stores and indexes for efficient similarity searches. Unlike traditional databases that rely on exact matching, VectorDBs quickly retrieve semantically or contextually similar items. This capability makes them essential for applications like semantic search, recommendation systems, and large-scale multimedia retrieval.

1.2 What is “Embedding”?

An embedding is a fixed-length, high-dimensional vector that numerically represents data (like text or images). Machine learning models, typically neural networks, create embeddings by mapping inputs into a vector space where similar items are positioned close together.

For this project, image embeddings are generated using the ResNet18 model, producing a 512-dimensional vector for each image. These vectors are stored in the vector database and compared using similarity metrics like cosine similarity. This enables efficient searches for visually similar images, making the quality of the embedding crucial for the system's overall accuracy.

These embedding vectors serve as the core data stored in the vector database. Once embedded, images can be compared using distance or similarity metrics such as Euclidean distance or cosine similarity, enabling efficient **k-nearest neighbor (k-NN)** search. As a result, the quality of the embedding directly affects the accuracy and effectiveness of similarity search within the system.

1.3 How do we measure similarity between vectors?

To measure how “close” or similar two vectors are in a vector space, we use mathematical similarity or distance functions. Common choices include:

- **Euclidean Distance** — the straight-line (L2) distance between vectors in multidimensional space.
- **Cosine Similarity** — the cosine of the angle between two vectors; this focuses on orientation rather than magnitude.

- **Manhattan Distance** — the sum of absolute differences along each dimension (L1 norm).
- **Dot Product** — the inner product of vectors; when vectors are normalized, this is equivalent to cosine similarity.

You can read more about these metrics in this [link](#).

1.4 Vector Databases vs. Traditional SQL Databases

SQL databases store **structured data** in tables composed of rows and columns, where each column has a predefined schema and data type. Querying is typically performed using **exact matching or range-based predicates** (e.g., `WHERE age > 20, name = 'Ali'`). These systems are optimized for **deterministic queries** and **transactional workloads** (OLTP), where correctness, consistency, and exactness are critical.

In contrast, vector databases store **high-dimensional numerical vectors** that represent embedded forms of unstructured data such as images, text, or audio. Queries are not based on exact equality but on **similarity** in a metric space. Instead of asking “*Which rows match this condition?*”, vector databases answer “*Which vectors are closest to this query vector?*”.

1.5 Indexing in Databases

1.5.1 What Is Indexing in Databases?

Indexing is a core technique in database systems for reducing the time complexity of search operations. Without an index, querying a dataset of size n requires a full scan with $O(n)$ time complexity. An index introduces an auxiliary data structure that restricts the search space, trading additional storage and preprocessing time for faster queries. Formally,

given a dataset D and a query Q , indexing aims to minimize the number of elements in D that must be examined to answer Q .

In traditional databases, data is low-dimensional and often ordered, allowing structures such as balanced trees or hash tables to achieve $O(\log n)$ or even constant-time lookups. These assumptions break down for vector data, where high dimensionality prevents meaningful total ordering and similarity is defined by distance functions rather than exact matches. As a result, classical indexes degrade toward linear-time performance for vector similarity search, motivating specialized indexing techniques.

1.5.2 Indexing in Vector Databases

Vector database indexing targets efficient k-nearest neighbor (k-NN) search in high-dimensional metric spaces. A naïve k-NN search computes distances to all vectors, requiring $O(n \cdot d)$ time, where d is the vector dimension. Vector indexes instead reduce the number of distance computations by retrieving a small candidate set, achieving sub-linear expected query time at the cost of approximate results. Common approaches include Locality Sensitive Hashing (LSH), which hashes similar vectors into the same buckets, graph-based methods such as HNSW, which traverse proximity graphs, and tree-based random projection methods such as Annoy.

Without any indexing, similarity search degenerates to brute-force comparison against all N vectors, which is $O(N)$ per query and becomes impractical for large-scale datasets.

1.5.3 Indexing in Vector Databases (Methods)

1.5.3.1 Hash-Based Indexing: Locality Sensitive Hashing (LSH)

Locality Sensitive Hashing (LSH) is an approximate nearest neighbor search technique that hashes similar high-dimensional data points into the same "buckets" with high probability. Unlike traditional hashing, which separates data, LSH uses functions designed so that the closer two items are (under a metric like cosine distance), the more likely they are to share a hash value, enabling fast retrieval.

For image similarity, LSH is applied not to raw pixels but to numerical embeddings generated by models like ResNet. This transforms the image search into a fast, approximate search in vector space, where querying involves hashing a new image's embedding and checking only the small subset of pre-processed images in its matching buckets.

For a rigorous theoretical treatment of Locality Sensitive Hashing, you are encouraged to consult **Chapter 3 (Finding Similar Items)** of *Mining of Massive Datasets* by Leskovec, Rajaraman, and Ullman. Supplementary lecture videos and course materials are available through the official course page at the provided [link](#), which offers an in-depth discussion of similarity measures, shingling, min-hashing, and LSH foundations.

1.5.3.2 Tree-Based Indexing: Random Projection Trees (Annoy)

Annoy (Approximate Nearest Neighbors Oh Yeah), published by Spotify, uses multiple random projection trees to index vectors. Each tree recursively splits the data using randomly chosen hyperplanes, resulting in a hierarchical partition of the vector space.

Building the index requires $O(n \cdot \log n)$ time per tree. During querying, the tree is traversed in $O(\log n)$ time to identify candidate vectors, followed by exact distance computation on a small subset. This approach significantly reduces the practical query time compared to brute-force search, while maintaining linear space complexity with respect to the number of vectors.

Annoy is particularly effective for static datasets with heavy read workloads.

1.5.3.3 Graph-Based Indexing: HNSW

Graph-based indexing methods such as Hierarchical Navigable Small World (HNSW) represent vectors as nodes in a proximity graph. Each node is connected to a limited number of nearby nodes, enabling efficient greedy traversal.

HNSW achieves an expected query time of $O(\log n)$ by navigating from coarse graph layers to finer ones. Index construction also runs in $O(n \cdot \log n)$ expected time, with linear space overhead. Despite its approximate nature, HNSW often achieves near-exact accuracy in practice.

2. Tasks:

2.1. Designing the Storage Structure and Persistence Layer

Design a data structure to store vectors along with their unique IDs and associated metadata.

The system must support **persistent storage on disk**, ensuring that vectors and metadata remain available across program executions. In-memory data structures (e.g., dynamic arrays or lists) may be used as working buffers, but the authoritative data representation must be stored on disk.

Acceptable storage formats include binary array formats, serialized objects, or custom binary layouts. The design should explicitly document:

- How vectors are stored and loaded from disk?
- How IDs and metadata are associated with vectors?

All design decisions must be documented in both the technical report and the README file.

2.2. Implementing CRUD Operations

- **Create:** insert a new vector
- **Read:** retrieve a vector by its ID
- **Update:** modify an existing vector
- **Delete:** remove a vector by its ID

2.3. Implementing the Brute-Force kNN Algorithm

You will work with a pre-embedded image dataset derived from the [Caltech-101](#) dataset using a pre-trained ResNet18 model.

Extracted image embeddings were persisted to disk as NumPy arrays to enable efficient reuse during exact and approximate nearest neighbor experiments, avoiding repeated feature extraction overhead.

The embedded dataset can be downloaded from this [link](#).

Optionally, you may generate and use embeddings for the [Caltech-256](#) or any other datasets for extended evaluation or bonus experiments.

- Implement an image preprocessing pipeline compatible with ResNet18 requirements
- Convert images to 512-dimensional feature vectors using pre-trained ResNet18
- Compute similarity/distance between a query vector and all vectors in the database
- Provide benchmark analysis including:
 - time analysis in an Excel file (similar format to previous lab submissions)
 - time complexity order (Big-O notation)
 - Comparison between cosine similarity and Euclidean distance

2.4. Implementing Approximate Nearest Neighbor (ANN) Search

Implement **Locality Sensitive Hashing (LSH)** as the **required** approximate nearest neighbor algorithm for this project. The implementation must support similarity search over high-dimensional image embeddings and demonstrate sub-linear query performance in practice.

The following algorithms are **optional bonus implementations** and will be considered for additional credit if correctly implemented and analyzed:

1. HNSW / NSW (Graph-Based Indexing)
2. Annoy-style Random Projection Trees

Bonus implementations must include performance and accuracy comparisons against both brute-force search and LSH.

2.5. Graphical User Interface (GUI)

You're supposed to develop a minimal graphical user interface to present your project during submission review and demonstrate the successful results you achieved in your db.

Required Features:

- Ability to select and import an image outside the database and show (scroll) the most similar ones in the database
- Display an **embedding scatter plot** or other appropriate **scatter plots** to visualize the final outcome of your database using convenient libraries like: Matplotlib, Seaborn, Tensorflow.

Optional Features:

- Ability to select an image category from the database and efficiently retrieve the most similar images.

2.6. Optional Task: Concurrency and Atomicity (Bonus)

Extend the vector database to support **concurrent access** from multiple threads or processes, ensuring **atomicity and consistency** during simultaneous read and write operations. Students may assume a shared-memory model and use synchronization mechanisms such as mutexes, locks, or atomic operations to protect critical sections. Concurrent CRUD operations must not lead to data corruption or inconsistent states. A brief explanation of the concurrency model, synchronization approach, and performance impact should be included in the report.

2.7. Report & Documentation

- Include a *readme.md* file containing clear instructions on how to run/compile your project. Also, write a brief explanation of what your project does and how it works
- Submit a technical report (Word/LaTeX) mentioning the speed and accuracy of each algorithm implemented and a comparison between them. Include research findings and explanations of how the algorithms work.

Submission Requirements

- GitHub repository containing complete source code and a *readme.md* file
- Technical report (Word or LaTeX format)