VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Data Structures & Algorithms (Extended) – CO200B

Text Deduplication

Instructor:	Lê Thành Sách
Students:	Ngô Kỳ Nam - 2212135
	Bùi Ngọc Phúc - 2312665

HO CHI MINH City, December, 2025

Contents

1 Objective **3**

2 Theoretical Basis **4**
2.1 SimHash . . . 4
2.1.1 Concept and Core Idea . . . 4
2.1.2 Working Principle . . . 4
2.1.3 Advantages and Disadvantages . . . 4
2.1.4 Typical Applications . . . 5
2.1.5 Conclusion . . . 5
2.2 MinHash . . . 5
2.2.1 Concept and Core Idea . . . 5
2.2.2 Working Principle . . . 6
2.2.3 Advantages and Disadvantages . . . 6
2.2.4 Typical Applications . . . 7
2.2.5 Conclusion . . . 7
2.3 FAISS . . . 7

3 Dataset creation & Preprocessing **8**
3.1 Creation . . . 8
3.2 Preprocessing . . . 9

4 System Design **10**
4.1 Overall pipeline . . . 10
4.2 Data Integration . . . 10
4.3 De-duplication & Index creation . . . 10
4.4 Searching . . . 11

5 Implementation Technique **12**
5.1 FAISS . . . 12
5.2 MinHash . . . 12
5.3 SimHash . . . 13

6 Evaluation and Method Comparison **14**
6.1 Objectives and Scope . . . 14
6.2 Experimental Setup . . . 14
6.3 Evaluation Metrics . . . 14
6.4 Experimental Results . . . 15
6.4.1 Memory Efficiency . . . 15
6.4.2 Time Efficiency . . . 16
6.4.3 Accuracy (Hitrate@k) . . . 17
6.4.4 Summary . . . 17

7 Future Improvements **18**

List of Figures

1 Overall application pipeline 10
2 Data integration flow chart 10
3 De-duplication flowchart 11
4 Search Module flowchart 11
5 Memory Usage for Indexing (Lower is Better) 15
6 Indexing Time (Left) and Search Time per Query (Right) Comparison 16
7 Hitrate @ k Comparison (Higher is Better) 17

1 Objective

The objective of this project is to develop a system capable of detecting and eliminating duplicate or near-duplicate texts within large-scale document collections. The system will employ deep learning techniques for content feature extraction, representing each text as a feature vector. Subsequently, various hashing methods (Hash Table, Bloom Filter, SimHash, MinHash) will be implemented and compared with FAISS to evaluate their effectiveness in identifying similar texts on real-world datasets.

Github: https://github.com/Kriss-Nevile/Extended-DSA
Google Colab: https://colab.research.google.com/drive/1wV3jP5VcRc62NNPZAkEbJgywI134EAT3?usp=drive_link

2 Theoretical Basis

2.1 SimHash

2.1.1 Concept and Core Idea

SimHash (Similarity Hashing) is a feature hashing technique designed to detect *near-duplicate* or highly similar objects rather than exactly identical ones. Unlike traditional cryptographic hash functions (e.g., MD5, SHA-256), which completely change their output even when the input differs by a single character, SimHash is built to be *content-sensitive*, meaning that two similar texts will produce almost identical hash codes.

Essentially, SimHash is a form of **Locality Sensitive Hashing (LSH)** for the *Cosine similarity* measure, proposed by Moses Charikar (2002). This technique maps each text document into a fixed-length **binary feature vector** (typically 64 or 128 bits), such that the Hamming distance between two vectors reflects the degree of content similarity between the corresponding documents.

2.1.2 Working Principle

Assume that a document is represented by a set of features $\{t_1, t_2, ..., t_n\}$, where each feature is assigned a weight w_i (e.g., based on TF-IDF frequency). The SimHash algorithm operates through the following steps:

1. **Preprocessing:** Normalize the text (convert to lowercase, remove punctuation, stopwords, etc.) and split it into tokens (words, n-grams, or other features).
2. **Feature Hashing:** For each token t_i, apply a hash function $h(t_i)$ to obtain a k-bit binary string b_i.
3. **Weighted Summation:** Initialize a vector $V = [0, 0, ..., 0]$ of length k. For each token:
   - If the j-th bit of b_i is 1, then $V_j = V_j + w_i$
   - If the j-th bit of b_i is 0, then $V_j = V_j - w_i$
4. **SimHash Generation:** For each bit position j:

$$\text{SimHash}_j = \begin{cases} 1, & \text{if } V_j > 0 \\ 0, & \text{if } V_j \leq 0 \end{cases}$$

   The result is a k-bit binary vector representing the entire document.

5. **Similarity Comparison:** Two documents are compared using the *Hamming distance* between their SimHash values. A smaller distance indicates higher similarity.

Example: If two 64-bit SimHash codes differ by only 3 bits, they can be considered near-duplicates.

2.1.3 Advantages and Disadvantages

Advantages:

- Very fast computation, suitable for large-scale datasets (e.g., web-scale).
- Compact storage: each document is represented by only a few bytes.

- Can detect near-duplicate texts even with small modifications (e.g., reordering, synonym replacement, or sentence insertion).
- Easily integrates with approximate search frameworks such as **Locality Sensitive Hashing (LSH)** or **FAISS** for efficient similarity retrieval.

Disadvantages:

- Accuracy depends heavily on feature selection and weighting (e.g., TF-IDF, frequency-based features).
- Cannot effectively capture deep semantic changes in meaning.
- Less reliable for very short or noisy texts.

2.1.4 Typical Applications

SimHash is widely used in various large-scale data processing systems:

- **Near-duplicate document detection:** Used by Google to eliminate duplicated web pages in its search index.
- **Plagiarism detection:** Identifying similar or copied text segments.
- **Spam or comment deduplication:** Filtering repetitive user-generated content.
- **Fast similarity search:** Retrieving similar documents using Hamming distance-based comparisons.

2.1.5 Conclusion

SimHash is an efficient feature hashing technique that maps textual data into a compact binary space while preserving similarity relationships. Thanks to its low computational complexity and scalability, SimHash has become a standard tool in large-scale duplicate detection systems, especially when combined with deep learning models that generate semantic feature embeddings for textual content.

2.2 MinHash

2.2.1 Concept and Core Idea

MinHash (Minimum Hashing) is a feature hashing technique designed to efficiently estimate the similarity between two sets without having to compare all of their elements directly. The core idea of MinHash is to generate a compact *fingerprint* for each set, such that the probability of two fingerprints being identical is exactly equal to the **Jaccard similarity** between the two original sets.

Given two sets A and B, the Jaccard similarity is defined as:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

MinHash allows this similarity value to be approximated by comparing only a small number of hash values, rather than explicitly computing the intersection and union of large sets, thereby significantly reducing time and memory costs for large-scale systems.

2.2.2 Working Principle

MinHash is based on the idea of using multiple random hash functions to map each element of a set to an integer value, and then keeping only the minimum hash value for each function.

1. **Set Construction:** Represent each document or object as a set of features (e.g., words, tokens, or n-grams).
2. **Define Hash Functions:** Create k independent hash functions $h_1, h_2, ..., h_k$.
3. **Compute MinHash Values:** For each hash function h_i, compute:

   $$\text{MinHash}_i(S) = \min_{x \in S} h_i(x)$$

   The resulting **signature vector** for the set S is:

   $$\text{Sig}(S) = [\text{MinHash}_1(S), \text{MinHash}_2(S), ..., \text{MinHash}_k(S)]$$

4. **Similarity Estimation:** The similarity between two sets A and B can be estimated as:

   $$\widehat{J}(A, B) = \frac{1}{k} \sum_{i=1}^{k} I[\text{MinHash}_i(A) = \text{MinHash}_i(B)]$$

   where $I[\cdot]$ is the indicator function. As k increases, the estimated similarity $\widehat{J}(A, B)$ converges to the true Jaccard similarity $J(A, B)$.

Example: If two sets share 80% of their elements, then roughly 80% of their MinHash values will also be identical.

2.2.3 Advantages and Disadvantages

Advantages:

- Enables fast estimation of Jaccard similarity without scanning full datasets.
- Highly efficient for very large datasets (web-scale, logs, documents, images).
- Can be combined with **Locality Sensitive Hashing (LSH)** to accelerate approximate similarity searches.
- Supports extensions such as **Weighted MinHash** for handling sets with weighted or frequency-based elements.

Disadvantages:

- Accuracy depends on the number of hash functions k — too few functions lead to high variance in similarity estimation.
- Applicable primarily to the *Jaccard similarity* measure; adaptations are required for other similarity metrics.
- Generating and storing multiple hash functions can be memory-intensive for very large-scale data.

2.2.4 Typical Applications

MinHash has been widely adopted in many systems and research domains, especially when the goal is to compare or cluster objects based on overlapping feature sets:

- **Near-duplicate document detection** in large text corpora (e.g., Google News, Wikipedia).
- **Document similarity search** based on the Jaccard similarity of keywords or n-grams.
- **Recommendation systems** — grouping users or products with similar behavior profiles.
- **Log and event analysis** — identifying repeating patterns in system activity.
- **Web and social media analytics** — clustering posts, comments, or hashtags with similar content.

2.2.5 Conclusion

MinHash is an efficient and elegant technique for quickly estimating the similarity between large sets. When combined with **Locality Sensitive Hashing (LSH)**, it enables fast and scalable approximate nearest-neighbor searches. By representing each set with a compact signature vector, MinHash drastically reduces computational cost while maintaining high accuracy, making it a standard tool for large-scale search, deduplication, and similarity detection systems.

2.3 FAISS

FAISS (Facebook AI Similarity Search)[1] is a library developed by Meta AI for efficient similarity search and clustering of dense vectors. It is designed to scale from small datasets to collections with billions of vectors, including cases where the data exceeds system memory. Instances are represented as vectors, identified by integers, and compared using standard similarity measures. FAISS is implemented in C++ with wrappers to many other languages.

In particular, its primary goal is to solve the Approximate Nearest Neighbor Search (ANNS) problem, which aims to retrieve the closest vectors to a given query vector without performing exhaustive pairwise comparisons. This task is central to large-scale applications such as recommendation systems, information retrieval, and machine learning pipelines, where data often consist of millions or billions of high-dimensional embeddings.

Given a set of vectors $\{x_1, x_2, \ldots, x_n\}$ in $\mathbb{R}^d$, FAISS constructs an in-memory data structure known as an *index*. Once built, the index allows efficient evaluation of queries of the form

$$i = \arg\min_i \|x - x_i\|,$$

where $\|\cdot\|$ denotes the Euclidean (L2) distance. The search operation identifies the nearest neighbor (or multiple neighbors) of a query vector $x \in \mathbb{R}^d$. In FAISS, vectors are added to the index through an `add()` method, and their dimensionality is fixed. The system supports not only nearest neighbor search but also retrieval of the top-k closest vectors, batch processing of multiple queries for improved efficiency, and range search operations that return all vectors within a specified distance threshold.
FAISS can also perform maximum inner product search,

$$i = \arg\max_i \langle x, x_i \rangle,$$

and supports additional distance metrics such as L1 and L_∞. Beyond in-memory search, indexes can be stored on disk, and the library can handle binary as well as floating-point vectors. Many index types allow precision to be traded for speed, for instance by returning approximate neighbors to achieve significant gains in computational efficiency and memory usage.

The library provides a variety of index types, each reflecting different trade-offs among search time, accuracy and memory. Baseline indexes perform exact search, while compressed-domain methods such as binary representations and quantization allow searches at scale by reducing storage and computation costs, often at the expense of precision. Additionally, graph-based indexes such as HNSW and NSG build efficient structures on top of raw vectors to accelerate retrieval.[2]

3 Dataset creation & Preprocessing

3.1 Creation

We will be collecting our data based on several criteria. First off it should have samples of various different lengths, so that it we can build and evaluate the de-duplication pipeline on all kinds of data, thus making it more robust. We came up with the following partition for sentences length in our dataset, let s be the length of a particular data sample:

- Short ($s < 10$) (10%)
- Medium ($10 \leq s < 20$) (15%)
- Long ($20 \leq s < 200$) (70%)
- Very Long ($200 \leq s$) (5%)

Secondly we will want some form of ground truth in our dataset, that is samples with a corresponding positive or negative samples.

Following these criteria, We will be collecting and integrating data from 4 different sources, that is:

- Quora duplication dataset
- CNN Daily mail
- Amazon reviews
- STS-B

In which **Quora duplication** and **STS-B** contains the ground truth that we can use to calculate the accuracy of our search algorithm. Below is the recap of our synthesized, with a total of 87426 rows.

Name	Length Type	Count	Is Ground Truth
Quora duplication dataset	Medium	9398	✓
CNN Daily Mail	Long	31000	✗
CNN Daily Mail	Very Long	4419	✗
Amazon Reviews	Long	30955	✗
STS-B	Short	12354	✓

Table 1: Dataset partition

3.2 Preprocessing

In this pipeline we're considering 3 different embedding models: `all-minilm-l6-v2`, `bge-base-en-v1.5` and `e5-base-v2`. Since these models can only process up to a certain amount of tokens, we need to limit the samples in our dataset. To achieve this, we perform a sentence-splitting procedure on each samples in the dataset, which means the samples will be cut into chunks of full sentences instead of being cut off mid-word, this helps better preserve the semantic for each samples and makes the resulting dataset more coherent in general. We will also flatten the dataset into only 1 text column for the convenience of the search algorithm later on.

We will create separate dataset tailored to each embedding model, based on their max token length and their respective tokenizers. The table belows summarizes these datasets.

Model Name	Max Token Length	Sample Count
✗ (Base dataset)	✗	87426
`all-minilm-l6-v2`	256	123565
`bge-base-en-v1.5`	512	115254
`e5-base-v2`	512	115254

Table 2: Model-tailored Datasets

Q&A

Question: Why do we even need Long and Very Long samples if we end up splitting our data anyway?

Answer: Long and Very Long samples still matter because they preserve the full semantic context before splitting. Chunking inherits meaning from the original long sample, allowing us to perform effective de-duplication on the dataset.

4 System Design

4.1 Overall pipeline

We design our de-duplication application with several key capabilities in mind. First, it must allow users to ingest data into the pipeline using any embedding model of their choosing. The system will automatically split the provided dataset into correctly sized samples. Second, users can select the de-duplication method they wish to apply, after which the system returns a list of representative samples. A supporting tool is also included to inspect which original samples correspond to each representative. Once the dataset has been de-duplicated, users can perform similarity search on the resulting samples. Based on these requirements, we propose the following pipeline for our de-duplication application.

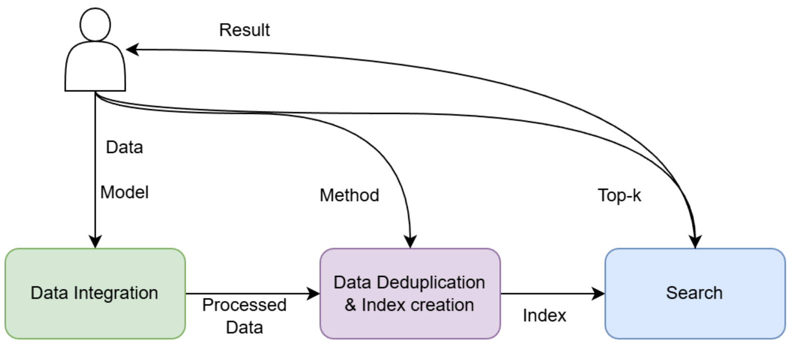

Figure 1: Overall application pipeline

4.2 Data Integration

The user provide the dataset, it is then passed through a preprocessing module, which obtains each model's maximum token limit and tokenizer, then splits the data into samples that fit within that limit. The resulting dataset is then fed into the model to generate embeddings.

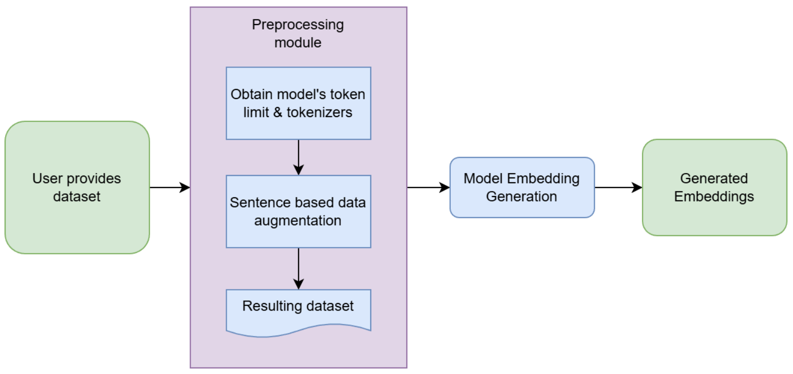
User provides dataset
Preprocessing module
Obtain model's token limit & tokenizers
Sentence based data augmentation
Resulting dataset
Model Embedding Generation
Generated Embeddings

Figure 2: Data integration flow chart

4.3 De-duplication & Index creation

The user first selects the desired de-duplication algorithm along with the threshold. After confirming these configurations, the application begins the de-duplication process and produces a dataset of representative texts, which the user can choose to download. This dataset is then used to create an index to enable better search performance, using the same algorithm as before. In addition, we provide a "view group members" option, after de-duplication is complete, the user can input the ID of any representative text, and the system will display all original samples that belong to that representative's group.

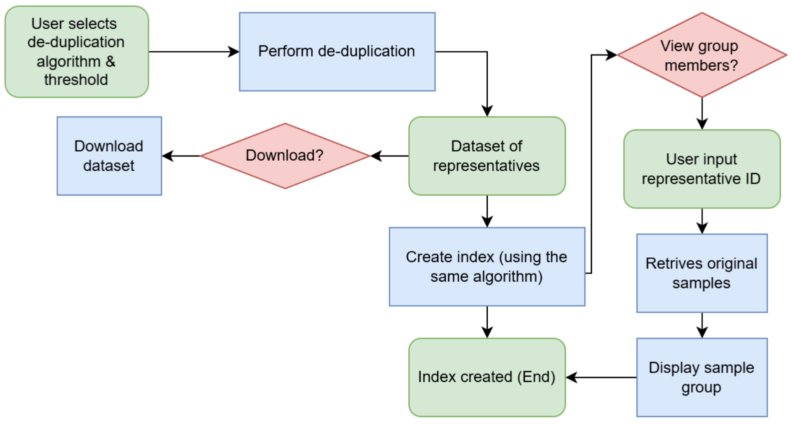
User selects de-duplication algorithm & threshold
Perform de-duplication
View group members?
Download dataset
Download?
Dataset of representatives
User input representative ID
Create index (using the same algorithm)
Retrives original samples
Index created (End)
Display sample group

Figure 3: De-duplication flowchart

4.4 Searching

This is relatively simple, the user input some text and choose the number of samples they want to retrieve (top-k). The input is encoded into a vector using the currently chosen embedding model, if using FAISS that embedding is directly used for similarity search, otherwise the embedding will be hashed using the same algorithm applied in the de-duplication phase, after that it will be used to perform top-k similarity search.

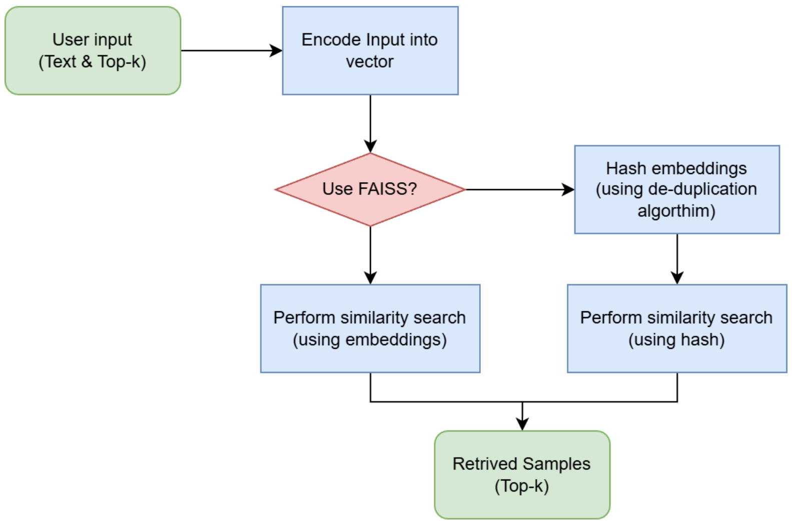
User input (Text & Top-k)
Encode Input into vector
Use FAISS?
Hash embeddings (using de-duplication algorthim)
Perform similarity search (using embeddings)
Perform similarity search (using hash)
Retrived Samples (Top-k)

Figure 4: Search Module flowchart

5 Implementation Technique

This section details the implementation of the three approximate nearest neighbor search techniques: FAISS LSH, MinHash, and SimHash. The implementation focuses on memory efficiency and query speed using Python and NumPy.

5.1 FAISS

For the baseline implementation, we utilized the Facebook AI Similarity Search (FAISS) library. We specifically employed the `IndexLSH` module, which implements Locality Sensitive Hashing.

- **Normalization:** Input vectors are first normalized using L2 normalization (`faiss.normalize_L2`) to ensure consistent distance calculations.
- **Configuration:** We configured the index with $n_{bits} = 256$.
- **Scoring:** The search returns the Hamming distance between the query and candidate vectors. We convert this to a similarity score using the formula:

$$Score = 1 - \frac{HammingDistance}{n_{bits}} \tag{1}$$

5.2 MinHash

The MinHash implementation is designed specifically for text data (short sentences) using a shingling approach. The process consists of two main components: the Encoder and the Indexer.

MinHash Encoder

The encoder converts raw text into MinHash signatures using the following steps:

1. **Tokenization:** Text is converted to lowercase, and non-alphanumeric characters are removed. Standard English stopwords are filtered out to reduce noise.
2. **Shingling:** The cleaned text is converted into 1-gram shingles. Each shingle is hashed into a 64-bit integer.
3. **Permutations:** We generate 256 signatures ($num\_perm = 256$) using vectorized hashing with large Mersenne primes for stability. The hash function for permutation i is defined as:

$$h_i(x) = (a_i \cdot x + b_i) \mod p \tag{2}$$

   where p is a large prime, and a_i, b_i are random coefficients.

MinHash Indexing (LSH)

To speed up retrieval, we implemented a custom LSH index using the *Bands and Rows* technique:

- **Banding:** The signatures are divided into bands (automatically calculated based on a threshold of 0.45).
- **Vectorized Hashing:** Instead of using standard hash maps, we use matrix multiplication to hash entire bands efficiently.
- **Sort-based Storage:** Candidates are stored in sorted arrays, allowing for fast binary search retrieval ($\mathcal{O}(\log N)$).

5.3 SimHash

The SimHash implementation is designed for dense vector embeddings. It projects high-dimensional vectors into binary fingerprints (bits) using Random Hyperplanes.

SimHash Encoder

The encoding process reduces the memory footprint by packing bits immediately:

1. **Random Projection:** We generate a random projection matrix R of shape (n_{bits}, D) drawn from a Gaussian distribution.
2. **Bit Generation:** For an input vector v, the fingerprint is calculated based on the sign of the dot product:

$$b_i = \begin{cases} 1 & \text{if } (v \cdot R_i) \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

3. **Packing:** The resulting bit array is packed into bytes using `np.packbits`, reducing memory usage by a factor of 8 compared to storing booleans.

SimHash Indexing

Similar to the MinHash implementation, we use a custom LSH structure without cosine reranking to maximize speed:

- **Structure:** The 256-bit signatures are split into 32 bands.
- **Query Strategy:** The system first identifies candidates that share at least one band with the query vector.
- **Ranking:** Final candidates are ranked purely by Hamming similarity (calculated via bitwise XOR and population count) to avoid expensive floating-point operations.

Configuration Used in Experiments

The experiments in this report use the following configuration:

- **SimHash:** `n_bits=256`, `n_bands=32`, `min_match_bands=3`, `max_candidates=500`.
- **MinHash:** `num_perm=256`, `ngram_size=1`, `threshold=0.45`, `min_band_matches=1`, `use_stopwords=True`.
- **FAISS LSH:** `IndexLSH` with `n_bits=256` and L2-normalization before indexing and querying.

6 Evaluation and Method Comparison

6.1 Objectives and Scope

The primary objective of this experiment is to evaluate the trade-offs between memory efficiency, processing speed, and retrieval accuracy among the three implemented techniques: FAISS LSH, SimHash, and MinHash. Specifically, we aim to determine:

- Which method minimizes Random Access Memory (RAM) usage during runtime.
- The computational cost required for indexing documents and querying nearest neighbors.
- The retrieval quality, measured by the ability to return relevant results within a specific tolerance threshold.

6.2 Experimental Setup

The experiments were conducted in a Python 3.x environment.

- **Dataset:** The evaluation utilizes a corpus of text documents preprocessed and converted into vector embeddings (for SimHash/FAISS) and shingled sets (for MinHash).
- **Parameters:**
  - **SimHash:** `n_bits=256`, `n_bands=32`, `min_match_bands=3`, `max_candidates=500`.
  - **MinHash:** `num_perm=256`, `ngram_size=1`, `threshold=0.45`, `min_band_matches=1`.
  - **FAISS LSH:** `IndexLSH` with `n_bits=256`

6.3 Evaluation Metrics

To assess performance comprehensively, we used the following metrics:

1. **Memory Usage (MB):** Peak memory consumption required to store the index structure.
2. **Indexing Time (s):** Total time taken to build the index structure from raw inputs.
3. **Search Time per Query (ms):** Average latency to process a single query and return candidates.
4. **Recall@k:** The proportion of relevant documents found in the top-k returned results. We evaluated recall at $k = 1, 5, 10$ to observe performance at different tolerance levels.

6.4 Experimental Results

6.4.1 Memory Efficiency

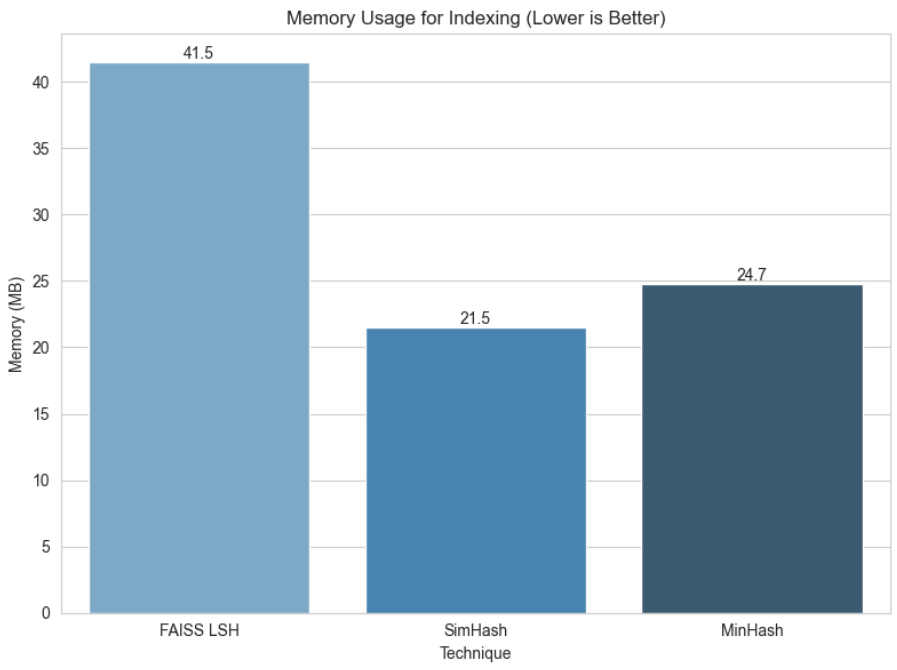

Figure 5: Memory Usage for Indexing (Lower is Better)

Figure 5 illustrates the memory footprint of each technique.

- **SimHash** proved to be the most memory-efficient method, consuming only **21.5 MB**. This is due to its compact bit-packing strategy (storing signatures as packed bytes).
- **MinHash** followed closely at **24.7 MB**.
- **FAISS LSH** consumed the most memory at **41.5 MB**, nearly double that of SimHash. This suggests that the FAISS wrapper maintains additional metadata or overhead for its internal index structures.

6.4.2 Time Efficiency

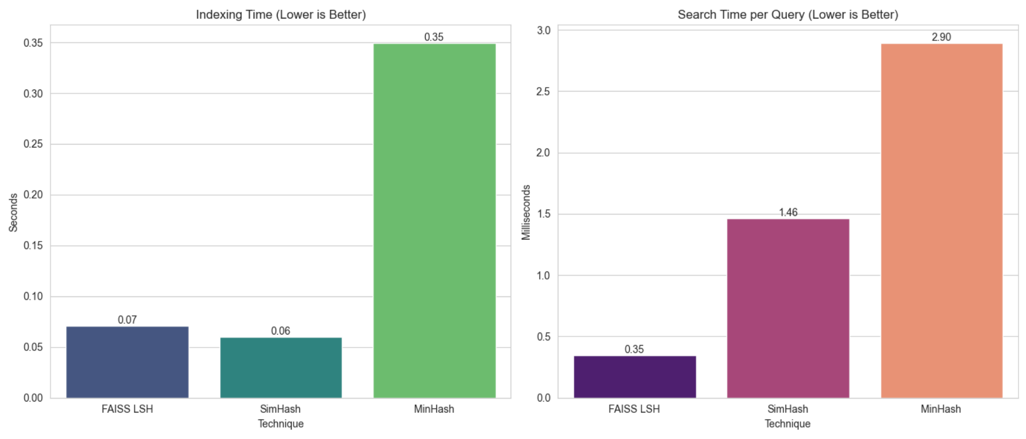

Figure 6: Indexing Time (Left) and Search Time per Query (Right) Comparison

The runtime performance is analyzed in two stages: Indexing and Searching.
Indexing Time:

- **SimHash (0.06s)** and **FAISS LSH (0.07s)** are extremely fast to build. The projection of vectors onto hyperplanes is computationally cheap.
- **MinHash (0.35s)** is significantly slower (approx. $5\times$ slower than SimHash). This is expected as computing 256 hash permutations for every document is CPU-intensive compared to vector dot products.

Search Time:

- **FAISS LSH** achieves the lowest latency at **0.35 ms** per query, benefiting from highly optimized C++ underlying implementations.
- **SimHash** performs reasonably well at **1.46 ms**.
- **MinHash** is the slowest at **2.90 ms**, likely due to the overhead of the Jaccard estimation and candidate verification steps in Python.

6.4.3 Accuracy (Hitrate@k)

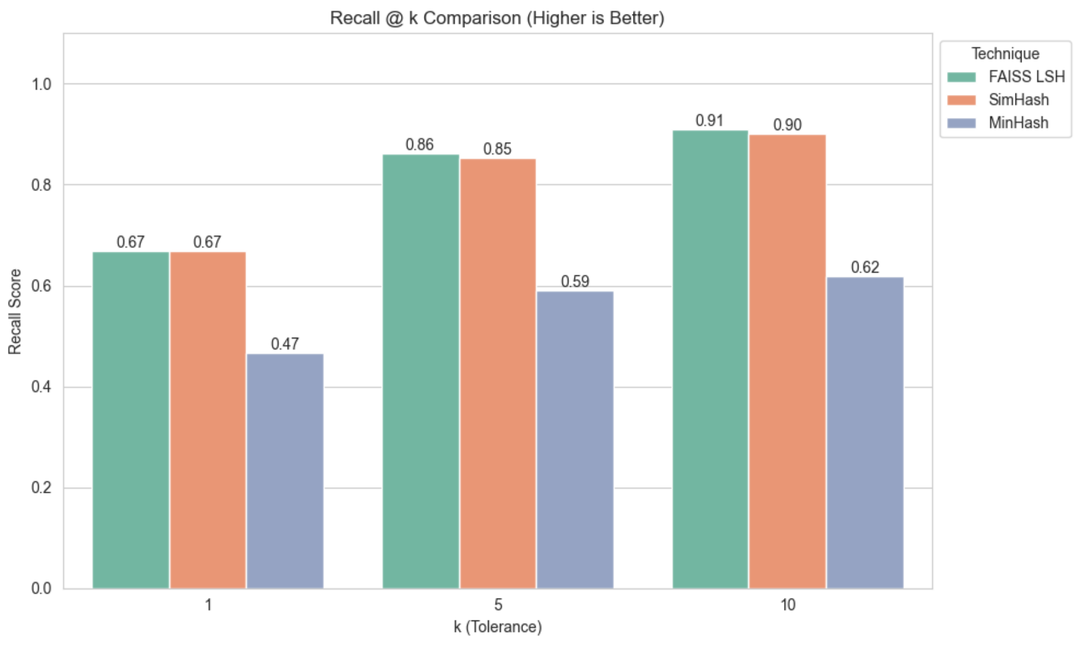

Figure 7: Hitrate @ k Comparison (Higher is Better)

The accuracy comparison is shown in Figure 7.

- **High Performance (FAISS & SimHash):** Both vector-based methods performed exceptionally well and identically. At $k = 10$, both achieved a hitrate of roughly **0.90–0.91**. This indicates that the LSH logic in our custom SimHash implementation aligns correctly with the standard FAISS library.
- **Lower Performance (MinHash):** MinHash showed significantly lower accuracy, with Hitrate@10 reaching only **0.62**.
- **Analysis:** The discrepancy suggests that for this specific dataset, semantic vector embeddings (captured by SimHash/FAISS) represent the document similarities better than the surface-level n-gram overlaps used by MinHash.

6.4.4 Summary

- For **speed and accuracy**, **FAISS LSH** is the preferred choice, though it uses more RAM.
- For **memory-constrained** environments, **SimHash** offers the best balance: lowest memory usage (21.5 MB), high accuracy (equal to FAISS), and acceptable speed.
- **MinHash** is less suitable for this specific embedding-based task due to lower accuracy and slower build times, though it remains useful for pure text-overlap detection.

7 Future Improvements

- Allow users to input different file types such as PDF or Word.
- Allow users to upload and merge multiple files into a single dataset.
- Introduce a UI for configuring pipeline parameters (token limits, chunk size, deduplication method, etc.).
- Add persistence so that user can retain the index and data that they uploaded

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] FAISS Documentation - Github Wiki. https://github.com/facebookresearch/faiss/wiki.