Bruno Francisco Martins da Silva

Staged Vector Stream Similarity Search Methods with an Application to Classified Ad Retrieval

Dissertação de Mestrado

Dissertation presented to the Programa de Pós–graduação em Informática, do Departamento de Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Marco Antonio Casanova

Rio de Janeiro
January 2024

Pontifícia Universidade Católica
do Rio de Janeiro

Bruno Francisco Martins da Silva

Staged Vector Stream Similarity Search Methods with an Application to Classified Ad Retrieval

Dissertation presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática. Approved by the Examination Committee:

Rio de Janeiro, January 24th, 2024

Bruno Francisco Martins da Silva

Graduated in Computer Science from the Federal Rural University of Rio de Janeiro (UFRRJ).

Bibliographic data

Silva, Bruno F. M.

Staged Vector Stream Similarity Search Methods with an Application to Classified Ad Retrieval / Bruno Francisco Martins da Silva; advisor: Marco Antonio Casanova. – 2024.

60 f: il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2024.

Inclui bibliografia

1. Informática – Teses. 2. Busca. 3. Indexação. 4. Similaridade. 5. Anúncios. 6. Redis. 7. HNSW. I. Casanova, Marco A.. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgments

I would like to express my gratitude to my Advisor, Prof. Marco Antonio Casanova, who brightly guided me through this research.

I would also like to thank my friend, João Pedro, who has shared with me all the tough and bright moments during this journey, my parents, Alzira de Jesus Martins da Silva and Marco Antonio da Silva, for their long-term support.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Abstract

Silva, Bruno F. M.; Casanova, Marco A. (Advisor). **Staged Vector Stream Similarity Search Methods with an Application to Classified Ad Retrieval**. Rio de Janeiro, 2024. 60p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A vector stream can be modeled as a sequence of pairs $((v_1, t_1) \ldots (v_n, t_n))$, where v_k is a vector and t_k is a timestamp such that all vectors are of the same dimension and $t_k < t_{k+1}$. The vector stream similarity search problem is defined as: "Given a (high-dimensional) vector q and a time interval T, find a ranked list of vectors, retrieved from a vector stream, that are similar to q and that were received in the time interval T". This dissertation first introduces a family of vector stream similarity search methods that do not depend on having the full set of vectors available beforehand but adapt to the vector stream as the vectors are added. The methods generate a sequence of indices that are used to implement approximated nearest neighbor search over the vector stream. Then, the dissertation describes an implementation of a method in the family based on Hierarchical Navigable Small World graphs. Based on this implementation, the dissertation presents a Classified Ad Retrieval tool that supports classified ad retrieval as new ads are continuously submitted. The tool is structured into a main module and three auxiliary modules, where the main module is responsible for coordinating the auxiliary modules and for providing a user interface, and the auxiliary modules are responsible for text and image encoding, vector stream indexing, and data storage. To evaluate the tool, the dissertation uses a dataset with approximately 1 million records with descriptions of classified ads and their respective images. The results showed that the tool reached an average precision of 98% and an average recall of 97%.

Keywords

Search; Indexing; Similarity; Ads; Redis; HNSW.

Resumo

Silva, Bruno F. M.; Casanova, Marco A.. **Metodos de Busca por Similaridade em Sequências Temporais de Vetores com uma Aplicação à Recuperação de Anúncios Classificados**. Rio de Janeiro, 2024. 60p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Uma sequência temporal de vetores ("vector stream") pode ser modelada como uma sequência de pares $((v_1, t_1) \ldots (v_n, t_n))$, onde v_k é um vetor e t_k é carimbo de tempo tais que todos os vetores são da mesma dimensão e t_k $< t_{k+1}$. O problema de busca por similaridade em sequências temporais de vetores é definido como: "Dado um vetor (de alta dimensão) v e um intervalo de tempo T, encontre uma lista ranqueada de vetores, recuperados de uma sequência temporal de vetores, que sejam similares a v e que foram recebidos dentro do intervalo de tempo T". Esta dissertação primeiro introduz uma família de métodos de busca por similaridade em sequências temporais de vetores que não dependem da sequência completa, mas se adaptam à medida que os vetores são incluídos na sequência. Os métodos geram uma sequência de índices, que são então usados para implementar uma busca aproximada do vizinho mais próximo na sequência temporal de vetores. Em seguida, a dissertação descreve uma implementação de um método da família baseado em Hierarchical Navigable Small World graphs. Utilizando esta implementação, a dissertação apresenta uma ferramenta de busca de anúncios classificados que oferece recuperação de anúncios à medida que usuários continuamente submetem novos anúncios. A ferramenta é estruturada em um módulo principal e três módulos auxiliares, sendo que o módulo principal é responsável por coordenar os módulos auxiliares e prover uma interface para o usuário, e os módulos auxiliares são responsáveis pela codificação dos textos e imagens em vetores, a indexação dos vetores, e o armazenamento dos textos, imagens e vetores. Por fim, para avaliar a ferramenta, a dissertação utiliza um conjunto de aproximadamente 1 milhão de registros com as descrições de anúncios classificados e suas imagens. Os resultados mostraram que a ferramenta atingiu uma precisão de 98% e um recall de 97%.

Palavras-chave

Busca; Indexação; Similaridade; Anúncios; Redis; HNSW.

Table of contents

List of figures

List of Abreviations

CNN – Convolutional Neural Network
LSTM – Long Short-Term Memory
RNN – Recurrent Neural Network
HNSW – Hierarchical Navigable Small World
IVFADC – Inverted File with Asymmetric Distance Computation

1 Introduction

This dissertation describes a family of methods, called *staged vector stream similarity search* methods, or briefly SVS, to help address the vector stream similarity search problem, defined as: “Given a (high-dimensional) vector q and a time interval T, find a ranked list of vectors, retrieved from a vector stream, that are similar to q and that were received in the time interval T”. The key observation is that SVS does not depend on having the full set of vectors available beforehand, but it adapts to the vector stream as the vectors are received. SVS generates a sequence of sets of indexed vectors and uses the indices to implement approximated nearest neighbor search over the vector stream.

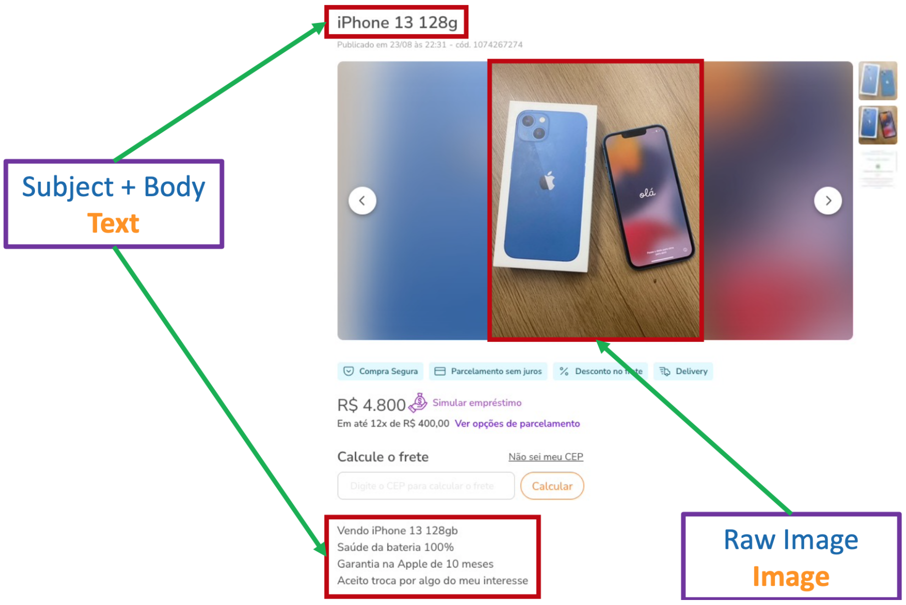


Figure 1.1: Example of a classified ad.

An instance of this problem arises in the context of a classified ad retrieval tool, where sellers can post classified ads, as in Figure 1.1, and buyers can search for products. The tool would combine text and content-based image retrieval (HAMEED; ABDULHUSSAIN; MAHMMOD, 2021; LI; YANG; MA, 2021), since product descriptions contain text and images. It might use separate high-dimensional vectors, created using Deep Learning techniques, to represent the text and images of an ad. Alternatively, the tool might transform the text and the images (or, in fact, any other type of media) of an ad into a single high-dimensional vector, as in cross-modal retrieval techniques (PEREIRA et al., 2014; ZENG; YU; OYAMA, 2020). In either case, the challenge lies in implementing approximated nearest neighbor search over high-dimensional vectors (JOHNSON; DOUZE; JEGOU, 2021; JéGOU; DOUZE; SCHMID, 2011; YANG et al., 2020). A second difficulty that the tool must face lies in that the set of classified ads is dynamic, in the sense that sellers continuously create new ads, often at a high rate, and ads may be short-lived, either because the product was sold, or because the seller withdraw the ad, or simply because the ad became obsolete for some reason. Hence, in conjunction, these two observations indeed lead to vector stream similarity search.

The dissertation first describes SVS. Briefly, it proposes to use a main memory cache C to temporarily store the vectors as they are received from the vector stream. When C becomes full or a timeout occurs, the current *stage* terminates and the vectors in C are indexed and stored in secondary storage. The net result is a sequence of indexed sets of vectors, each set covering a specific time interval. Hence, SVS is *incremental*, in the sense that it does not depend on having the full set of vectors available beforehand, but it adapts to the vector stream, and it can cope with an unlimited number of vectors.

Then, the dissertation presents two implementations of SVS: one is based on IVFADC - “Inverted File with Asymmetric Distance Computation” (JéGOU; DOUZE; SCHMID, 2011), and is called *staged IVFADC*; and another is based on HNSW – “Hierarchical Navigable Small World” graphs (MALKOV; YASHUNIN, 2020), as implemented in Redis[1], and is called *staged HNSW*. IVFADC and HNSW were chosen since they are well-known approximated vector similarity search methods.

Next, the dissertation describes two sets of experiments to assess the implementations. The experiments with staged IVFADC adopt the database and query descriptors from the INRIA Holidays images (JEGOU; DOUZE; SCHMID, 2008), and estimate the overhead of staged IVFADC against a non-staged implementation of IVFADC. The set of experiments with staged HNSW use a test dataset constructed from real data, and provides a more realistic comparison between a staged and a non-staged implementation.

Finally, to test SVS in practice, the dissertation includes a description of a proof-of-concept implementation of a classified ad retrieval tool based on the staged HNSW implementation to index the vector stream.

SVS was first introduced in (PINHEIRO et al., 2023). This dissertation clarifies the description of SVS, provides additional details about staged IV-FADC and related experiments, expands the experiments with staged HNSW, and details the proof-of-concept implementation.

[1]https://redis.io

The rest of the dissertation is organized as follows. Chapter 2 introduces background concepts and summarizes related work. Chapter 3 briefly summarizes some concepts used in the dissertation and reviews related work. Chapter 4 describes the tool used in this dissertation. Chapter 5 describes the experiments and compares the results. Finally, Chapter 6 presents the conclusions and directions for future research.

2
Background and Related Work

This section briefly summarizes some concepts used in the dissertation and reviews related work. It first covers neural network architectures to create vector representations of text and images used in the dissertation. Then, it reviews vector indexing methods, libraries and search engines, concluding with online vector similarity search methods.

2.1
Transformers

The Transformer is a model architecture eschewing recurrence and instead relying entirely on an attention mechanism to draw global dependencies between input and output. The architecture features an encoder-decoder structure, where the encoder is composed of a stack of identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. A residual connection is employed around each of the two sub-layers, followed by layer normalization. That is, the output of each sub-layer is $LayerNorm(x+Sublayer(x))$, where $Sublayer(x)$ is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of the same dimension.

The decoder is also composed of a stack of identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, residual connections are employed around each of the sub-layers, followed by layer normalization. Then the the self-attention sub-layer in the decoder stack is modified to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i (VASWANI et al., 2017).

2.2
Convolutional Neural Networks.

Convolutional Neural Networks (CNNs) are feedforward networks in that information flow takes place in one direction only, from their inputs

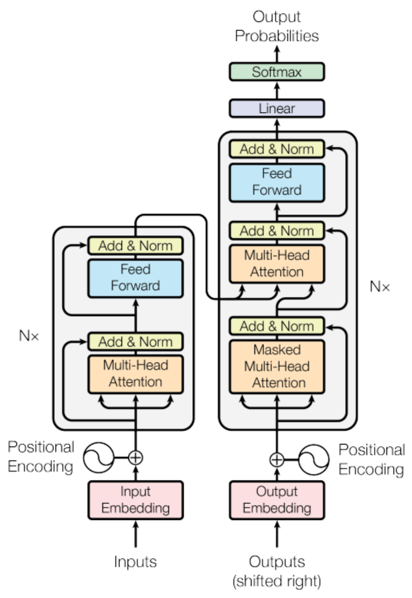


Figure 2.1: Transformers Architecture.

to their outputs. Just as artificial neural networks (ANN) are biologically inspired, so are CNNs. The visual cortex in the brain, which consists of alternating layers of simple and complex cells, motivates their architecture. CNNs architectures consists of convolutional and pooling (or subsampling) layers, which are grouped into modules. Either one or more fully connected layers, as in a standard feedforward neural network, follow these modules. Modules are often stacked on top of each other to form a deep model. Figure 2.2 illustrates the simplest design of the CNN architeture. Here an image is input directly to the network, and this is followed by several stages of convolution and pooling. Thereafter, representations from these operations feed one or more fully connected layers. Finally, the last fully connected layer outputs the class label.

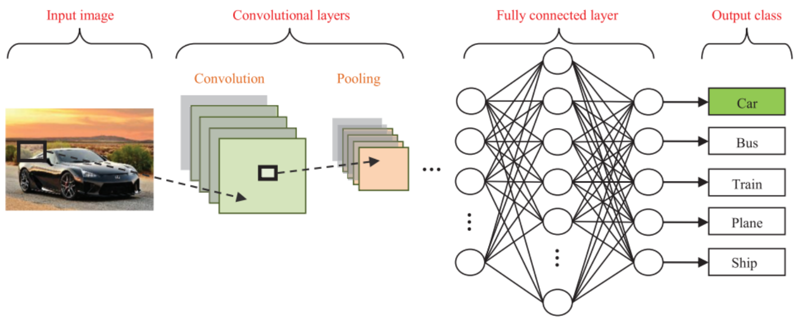


Figure 2.2: CNN Architecture.

2.3
Vector Indexing Methods, Libraries, and Search Engines

2.3.1
Offline Vector Similarity Search

Indexing Methods. Similarity search on large scale, high dimensional datasets is an essential feature of several Deep Learning applications (BENGIO; COURVILLE; VINCENT, 2013). Indeed, such applications represent objects as high-dimensional vectors and use vector similarity search to find relevant objects.

However, an exhaustive search of a set of nearest neighbors can be prohibitively expensive (BEYER et al., 1999) and traditional indexing strategies do not fare much better (JéGOU; DOUZE; SCHMID, 2011). Several algorithms (DATAR et al., 2004; GIONIS et al., 1999; MUJA; LOWE, 2009) tried to tackle the time complexity problem by looking for the nearest neighbor, with high probability instead of an exact search. However, storing the indexed vectors in the main memory still posed a serious limitation for large volumes of data.

The approach proposed in (JéGOU; DOUZE; SCHMID, 2011) circumvents these memory constraints by storing a short code in memory, obtained through product quantization, instead of the original vectors. This results in a time and memory-efficient solution for indexing vectors and performing an approximate nearest neighbor search. The basic idea is to cluster the vectors and use each cluster centroid to index all vectors that belong to that cluster. In particular, IVFADC (JéGOU; DOUZE; SCHMID, 2011) is an access method based on product quantization that has been implemented and successfully tested over billions of vectors. An implementation of product quantization that takes advantage of GPUs was also reported in (JOHNSON; DOUZE; JEGOU, 2021).

In more detail, IVFADC uses two quantizers, called a *coarse quantizer* and a *product quantizer*, and a set of inverted lists to index and query vectors. The coarse quantizer is used to determine which inverted list L each vector v should be added to, and the residual is passed through the product quantizer to generate the shortcode that is stored in L, together with the identifier of v. IVFADC is asymmetric because a query vector q is not quantized by the product quantizer. The coarse quantizer of q is used to determine which set of at most w inverted lists should be searched, and the distances between residuals and shortcodes are directly computed. The k nearest neighbor vectors are then returned. Note that w and k are parameters of the query, and the search is not exhaustive, since only the entries in the selected inverted lists are searched.

IVFFlat is a simplified version of IVFADC, which only uses the coarse quantizer and thereby has a faster index construction and requires less storage space. Furthermore, if the query vector comes from the vector dataset, IVFFlat can achieve a 100% recall.

In another direction, Malkov et al. (MALKOV; YASHUNIN, 2020) proposed the *Hierarchical Navigable Small World – HNSW* index for the approximate k-nearest neighbor search based on navigable small-world graphs with controllable hierarchy. The motivation behind HNSW is to improve the NSW model search complexity, which can be done through the analysis of the routing process. This process is divided in two phases: "zoom-out" and "zoom-in". A greedy algorithm starts in the "zoom-out" phase from a low degree node and traverses the graph simultaneously increasing the node's degree until the characteristic radius of the node links length reaches the scale of the distance to the query. However, before this happens, the average degree of a node can stay relatively small, which leads to an increased probability of being stuck in a distant false local minimum.

To adress this HNSW incrementally builds a multi-layer structure consisting of a hierarchical set of proximity graphs (layers) for nested subsets of the stored elements. The ideia is to separate the links according to the length scale into different layers and then search in a multilayer graph, as seen in Figure 2.3. In this only a needed fixed portion of the connections for each element is evaluated, independently of the networks size, allowing a logarithmic scalability.

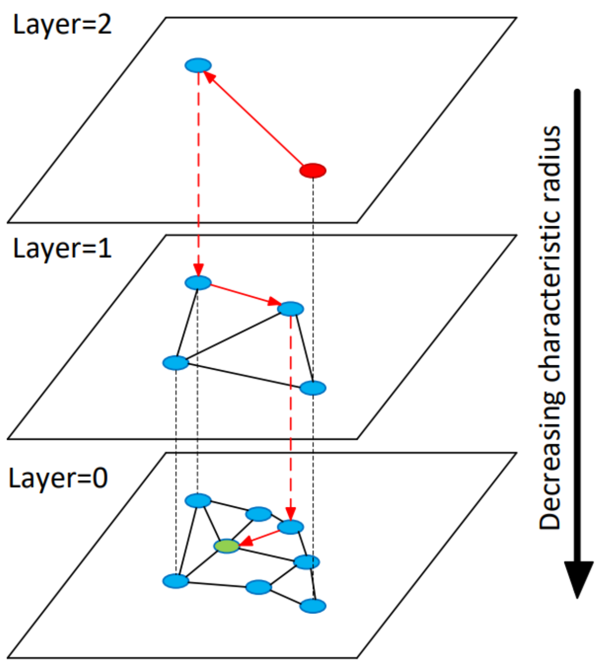


Figure 2.3: Illustration of HNSW idea

HNSW starts a search by randomly selecting an entry node from the top layer, then greedily traversing the graph to find the closest neighbor nodes to the entry node. After that, it continues to explore from the next layer using the found closest neighbors from the previous layer as new candidate nodes, repeating this process in each layer. In each step a list of k items is maintained. This list is updated by evaluating the neighborhood of the closest previously non-evaluated node in the list, until every node is evaluated. Here, HNSW exhibits its advantage compared to NSW algorithms, it allows discarting the candidates for evaluation that are further from the query than the furthest element in the list, thus avoiding bloating of search structures.

HNSW performs very well even on a large dataset, and can obtain a higher speedup than a quantization-based algorithm. When compared to NSW, it presents a complexity scaling not worse than logarithmic and outperforms NSW at any dataset size. Comparing it with product quantization based algorithms, it can achieve much higher accuracy, while offering a massive advance in search speed and much faster index construction, even though it requires significantly more RAM. It is important to note that HNSW spends a relatively long time building neighbor graphs. Graph storage is another bottleneck when the dataset is too large (FU et al., 2019).

Libraries. Several libraries have been implemented that offer vector indexing methods. They differ in the methods and the similarity metrics supported, as well as whether they are open source or not, offer a Python interface, and are stand-alone or run on a cluster, as summarized in Table 2.1.

FAISS[1] is an open-source Python library developed at Meta that offers several indexing methods and similarity metrics, including IVFADC and IVFlat. FAISS also has a multi-GPU implementation. ScaNN[2] is a similar library developed at Google. NGT[3] - Neighborhood Graph and Tree for Indexing High-dimensional Data was developed at Yahoo and implements a specific indexing method, with (NGTQ) or without quantization (NGT), with different similarity metrics.

Search Engines. Yang et al. (YANG et al., 2020) described PASE, a scheme for extending the index type of PostgreSQL that supports similarity vector search. PASE is used in an industrial environment and offers, among other options, IVFFlat and HNSW. The authors argued that IVFFlat is better for high-precision applications, such as face recognition, whereas HNSW performs better in general scenarios including recommendations and personalized advertisements. Milvus[4] is another example of a vector database offering similarity search. It supports, among others, IVFFlat and HNSW. Weaviate[5] is an open-source vector search engine that stores both objects and vectors, allowing for combining vector search with structured filtering with the fault-tolerance and scalability of a cloud-native database, all accessible through GraphQL, REST, and various language clients. Qdrant[6] and Elastic[7] are other examples of vector search engines.

Summary. Table 2.1 summarizes the main features of the vector indexing libraries and search engines mentioned in this brief review. A detailed comparison can be found at ANN-Benchmarks[8], a benchmarking environment for approximate nearest neighbor search algorithms.

Chapter 3 describes staged implementations based on IVFADC and HNSW, while Chapter 5 assesses the overhead of the staged implementations against non-staged, equivalent baselines.

2.3.2 Online Vector Similarity Search

Methods for batch similarity search of vectors were designed to cover the scenario where the complete set of vectors is known a priori. By contrast,

[1]https://github.com/facebookresearch/faiss/wiki/
[2]https://github.com/google-research/google-research/tree/master/scann
[3]https://morioh.com/p/8c38367453ae
[4]https://milvus.io/docs/index.md
[5]https://weaviate.io
[6]https://qdrant.tech
[7]https://www.elastic.co/what-is/vector-search
[8]http://ann-benchmarks.com/index.html

Table 2.1: A comparison of vector indexing libraries and search engines.

Tool	**Open Source**	**Multiple Similarity Metrics**	**Quantization**
FAISS	Y	Y*	Y
ScaNN	Y	Y*	Y
NGT	Y	Y*	Y
PASE	Y	Y	Y
Milvus	Y	Y	Y
Weaviate	Y	Y	
Qdrant	Y	Y	
Elastic	Y	Y	

(*) No support for cosine similarity.

methods for *online similarity search of vectors* were introduced to overcome this limitation.

Xu et al. (XU; TSANG; ZHANG, 2018) addressed the problem of creating quantization methods for databases that evolve. They described an online product quantization (online PQ) model that incrementally updates the quantization codebook to accommodate the incoming streaming data. Furthermore, the online PQ model supports both data insertions and deletions over a sliding window.

Liu et al. (LIU et al., 2020) also proposed an online, optimized product quantization model to dynamically update the codebooks and the rotation matrix.

Yukawa and Amagasa (YUKAWA; AMAGASA, 2021) proposed a method for updating the rotation matrix using SVD-Updating, which can update the singular matrix using low-rank approximations. By using SVD-Updating, instead of performing multiple singular value decompositions on a high-rank matrix, the authors showed how to update the rotation matrix by performing only one singular value decomposition on a low-rank matrix.

SVS follows a much simpler strategy. It generates a sequence of sets of indexed vectors, stores the indexes generated at each stage in secondary memory, and uses the stored indexes to process approximated nearest neighbor search over the high-dimensional vectors.

3
The Family of Staged Vector Stream Similarity Search Methods

This section briefly summarizes the methods utilized in this dissertation. It first presents the Staged Vector Stream Similarity Search methods and its algorithms, then it describes implementations based on IVFADC and finishes with Implementations based on HNSW.

3.1
Staged Vector Stream Similarity Search Methods

As a baseline, one may consider any vector similarity search method adapted to vector streams. Algorithm 1 summarizes, in pseudocode, the essence of a non-staged ingestion of a stream of vectors V (see Table 3.1 for the CreateIndex, Read, Clock, AdjustIndex, and Store procedures).

Algorithm 1 Non-staged ingestion of a stream of vectors V

```
1: procedure NS
2:     CreateIndex($I$)
3:     repeat
4:         Read($V; v$)
5:         $t \leftarrow$ Clock
6:         AdjustIndex($v, t, I$)
7:         Store($(v, t)$)
8:     until shutdown
9: end procedure
```

The exact details of an implementation of Algorithm 1 naturally depend on the index method chosen. However, independently of the method adopted, the index will grow unbounded since there is no limit on the number of vectors to be processed (recall that the vectors come from a stream). This is one of the problems that should be avoided.

The family of *staged vector stream similarity search methods*, or briefly *SVS*, refers to similarity search methods for vector streams with the following characteristics. SVS uses a main memory cache C to store the vectors as they are received from the vector stream. When C becomes full, or a timeout occurs, the current *stage* terminates and the vectors in C are indexed and stored in secondary storage. The net result is a sequence of indexed sets of vectors, each set covering a specific time interval. Hence, SVS does not depend on having the full set of vectors available beforehand, and it can cope with an unlimited number of vectors.

Table 3.1: SVS basic operations and auxiliary procedures.

Type	Operation
Basic	INGESTION of a stream of vectors, including indexing and storing the vectors in secondary storage
Basic	RETRIEVE vectors by similarity, and rank the retrieved vectors
Basic	DELETE a specific vector, given its identifier
Basic	MERGE time-adjacent indexes, if the indices become sparse
Aux	CLOCK returns the current wall clock value
Aux	READ a new vector from the stream
Aux	ADDCACHE adds a newly read vector to the cache
Aux	CREATEINDEX creates a new index
Aux	ADJUSTINDEX updates the index to register a vector
Aux	STORE moves data to secondary storage
Aux	RETRIEVEVECTORS performs an approximated nearest neighbor search to retrieve all vectors similar to a given vector

Table 3.1 lists the SVS basic operations and auxiliary procedures. Members of the SVS family differ basically on the exact vector indexing scheme they use. However, Pinheiro et al. (PINHEIRO et al., 2023) discussed two broad alternatives:

– *incremental*, when the index I is incrementally constructed, in main memory, as the vectors are added to the cache.

– *deferred*, when the index I is constructed, in main memory, at the end of each stage using all vectors in the cache.

In either case, I is persisted in secondary storage when the stage ends, and reinitialized for the next stage. This dissertation concentrates on the deferred alternative.

Algorithm 2 is a highly simplified description of the INGESTION operation in pseudocode, for the deferred alternative. It uses the auxiliary procedures as follows. When the cache becomes full, or a timeout occurs, CREATEINDEX is executed to create an index, I, required to index the vectors in C; STORE stores, in secondary storage, I with the time interval T it covers. STORE also moves to secondary storage each vector v in the cache C with the timestamp t when v was read.

To summarize, the main characteristics of Algorithms 1 and 2 are:

NS – Non-staged ingestion of a stream of vectors (Algorithm 1):

– Incrementally constructs a single index for all vectors in the stream.
– The overall cost is dominated by the cost of adjusting the index, since the number of vectors in the stream is not bounded.

ST – Staged ingestion of a stream of vectors (Algorithm 2):

– At the end of each stage, constructs an index for the vectors in the cache.
– At each stage, the cost of adjusting the index is bounded.
– At the end of each stage, the overhead is not negligible, since an index must be created using the vectors in the cache.
– At each stage, the index is specific to the vectors in the cache.

Finally, Algorithm 3 is again a highly simplified description of the RETRIEVE operation in pseudocode. Briefly, the RETRIEVE operation receives as input a query vector q and a time interval T and performs an approximated nearest-neighbor search over the stored vectors. For each index I whose interval intersects T, RETRIEVEVECTORS uses I to perform an approximated nearest neighbor search to retrieve from secondary storage all vectors indexed by I that are similar to q and whose timestamp falls in T, returning a list L_I of all such vectors. It combines the partial results in a single list L. Finally, it ranks the vectors in L by the distance to q and by timestamp. The RETRIEVE operation may also search the cache, if its time interval intersects T (not represented in Algorithm 3 for simplicity).

Algorithm 2 Staged ingestion of a stream of vectors V

```
1: procedure ST($timeout$)
2:     $C \leftarrow \emptyset$                                    ▷ cache
3:     $t_b \leftarrow$ CLOCK
4:     repeat
5:         READ($V; v$)                           ▷ read $v$ from stream $V$
6:         $t \leftarrow$ CLOCK
7:         ADDCACHE($(v, t), C$)
8:         $e \leftarrow$ (CLOCK $- t_b$)                  ▷ cache elapsed time
9:         if $C$ is full or $e > timeout$ then
10:            CREATEINDEX($C; I$)
11:            for each $(v, t) \in C$ do
12:                ADJUSTINDEX($v, I$)
13:                STORE($(v, t)$)
14:            end for
15:            $T \leftarrow (t_b,$ CLOCK$)$                   ▷ time interval
16:            STORE($(I, T)$)
17:            $C \leftarrow \emptyset$
18:            $t_b \leftarrow$ CLOCK
19:        end if
20:    until shutdown
21: end procedure
```

Algorithm 3 Retrieval of a ranked list of vectors L, given a query vector q and a time interval T

```
1: procedure RET($q, T$)
2:     $L \leftarrow \emptyset$
3:     for each index $I$ that covers $T$ do
4:         RetrieveVectors($q, I; L_c$)
5:         $L \leftarrow L \cup L_c$
6:     end for
7:     Rank $L$ by similarity to $q$ and by timestamp
8:     Return the ranked list
9: end procedure
```

3.2 Implementations based on IVFADC

IVFADC, outlined in Section 2.3, with some adjustments, would provide an implementation of the non-staged ingestion (see Algorithm 1). CreateIndex would construct a codebook I upfront from a *learning set* V_0 of vectors (Jegou et al. (JéGOU; DOUZE; SCHMID, 2011) used for the experiments a learning set with 100,000 images extracted from Flickr). AdjustIndex would then index each vector v in the stream against I, which reduces in IVFADC to finding the nearest centroid v_c in the coarse quantizer to v, using Euclidean distance, codifying the residual $r = v_c - v$ with the product quantizer into a code $q(r)$, and adding the ID of v and code $q(r)$ to the inverted list associated with v_c.

However, since the number of vectors in the stream is unknown, there is no limit on the size of the inverted lists that IVFADC uses to keep the indexed vector IDs and codes. Therefore, the inverted lists could be replaced by keeping the indexed vectors in a database. In fact, this is how PASE (YANG et al., 2020) implements IVFFlat in PostgreSQL.

IVFADC would also be an alternative to implement the staged Ingestion operation. At the end of each stage, CreateIndex would construct a different codebook I using the vectors in the cache, rather than using a training set. Then, AdjustIndex would index each vector v in the cache, that is, it would find the nearest centroid v_c in the coarse quantizer to v, using Euclidean distance, codifying the residual $r = v_c - v$ with the product quantizer into a code $q(r)$, and adding the ID of v and code $q(r)$ to the inverted list associated with v_c. However, contrasting with the discussion of the non-staged IVFADC, the size of the inverted lists is bounded, since it would depend on the size of the cache. Finally, Store would move the lists and the codebook to secondary storage. This implementation would use different codebooks in each stage, and would avoid the overhead of online product quantization methods. The disadvantage would be the overhead of constructing a new codebook at each stage, which might be reduced by sampling the vectors used in the clustering algorithm.

To summarize, the main characteristics of the IVFADC alternatives for the INGESTION operation are:

IVFADC-NS – IVFADC implementation of non-staged ingestion:

- Uses a fixed codebook, built upfront.
- Uses IVFADC to incrementally index all vectors in the stream.
- The overall cost is dominated by the cost of updating the inverted lists, since the codebook is fixed and created upfront from a training set of vectors.
- The inverted lists grow unbounded.

IVFADC-ST – IVFADC implementation of staged ingestion:

- At the end of each stage, constructs a different codebook and the inverted lists for the vectors in the cache.
- At the end of each stage, the overhead is not negligible, since a codebook and inverted lists are created for the vectors in the cache.
- At each stage, the codebook is specific to the vectors in the cache, which might increase recall.

3.3 Implementations based on HNSW

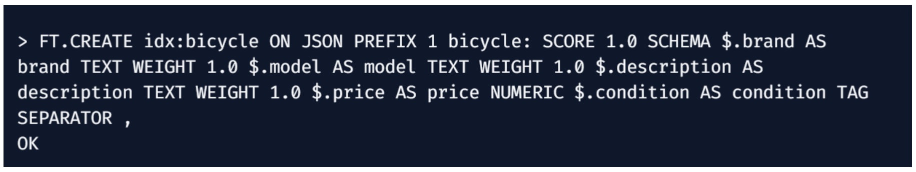
Redis [1] stands for Remote Dictionary Server. It is possible to use the same data types as in the local programming environment but on the server side. Similar to byte arrays, Redis strings store sequences of bytes, including text, serialized objects, counter values, and binary arrays.

Data is often unstructured, which means that it isn't described by a well-defined schema. There are many examples of unstructured data like texts, images, videos and others. An approach to dealing with unstructured data is to vectorize it. This means to map unstructured data to a flat sequence of numbers. Such a vector represents the data embedded in an N-dimensional space. Given a suitable machine learning model, the generated embeddings can encapsulate complex patterns and semantic meanings inherent in data. It is possible to use Redis as a vector database allowing to store vectors and the associated metadata within hashes or JSON documents, retrieve vectors and perform vector similarity searches

[1]https://redis.io

Each item within Redis has a unique key. All items live within the Redis keyspace. They can be scanned in the Redis keyspace via the SCAN command. SCAN returns a cursor position, allowing to scan iteratively for the next batch of keys until the cursor value 0 is reached. With that, other data structures (e.g., hashes and sorted sets) can be used as indexes, however the application would need to maintain those indexes manually. Redis can be turned into a document database by allowing you to declare which fields are auto-indexed.

The example in Figure 3.1 shows an FT.CREATE command that creates an index with some text fields, a numeric field (price), and a tag field (condition). The text fields have a weight of 1.0, meaning they have the same relevancy in the context of full-text searches. The field names follow the JSONPath notion. Each such index field maps to a property within the JSON document.



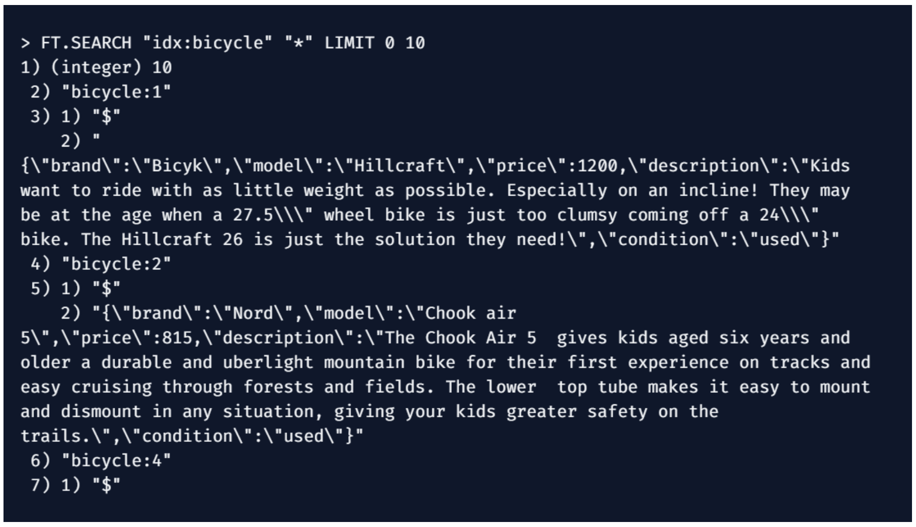
```
> FT.CREATE idx:bicycle ON JSON PREFIX 1 bicycle: SCORE 1.0 SCHEMA $.brand AS
brand TEXT WEIGHT 1.0 $.model AS model TEXT WEIGHT 1.0 $.description AS
description TEXT WEIGHT 1.0 $.price AS price NUMERIC $.condition AS condition TAG
SEPARATOR ,
OK
```

Figure 3.1: Create Index Example

As soon the FT.CREATE command is executed, the indexing process runs in the background. In a short time, all JSON documents should be indexed and ready to be queried. After an index is created, Redis Stack automatically indexes any existing, modified, or newly created JSON documents stored in the database.

For existing documents, indexing runs asynchronously in the background, so it can take some time before the document is available. Modified and newly created documents are indexed synchronously, so the document will be available by the time the add or modify command finishes. To validate that, you can use the FT.INFO command, which provides details and statistics about the index. Of particular interest are the number of documents successfully indexed and the number of failures.

To search the index for JSON documents, the FT.SEARCH command can be used. It allows any attribute defined in the SCHEMA to be searched. For example, Figure 3.2 uses a query to search for items with the index “bicycle”.



```
> FT.SEARCH "idx:bicycle" "*" LIMIT 0 10
1) (integer) 10
 2) "bicycle:1"
 3) 1) "$"
    2) "
{\"brand\":\"Bicyk\",\"model\":\"Hillcraft\",\"price\":1200,\"description\":\"Kids
want to ride with as little weight as possible. Especially on an incline! They may
be at the age when a 27.5\\\" wheel bike is just too clumsy coming off a 24\\\"
bike. The Hillcraft 26 is just the solution they need!\",\"condition\":\"used\"}"
 4) "bicycle:2"
 5) 1) "$"
    2) "{\"brand\":\"Nord\",\"model\":\"Chook air
5\",\"price\":815,\"description\":\"The Chook Air 5  gives kids aged six years and
older a durable and uberlight mountain bike for their first experience on tracks and
easy cruising through forests and fields. The lower  top tube makes it easy to mount
and dismount in any situation, giving your kids greater safety on the
trails.\",\"condition\":\"used\"}"
 6) "bicycle:4"
 7) 1) "$"
```

Figure 3.2: Search Index Example

More advanced techniques of searching can be used as well, like KNN. KNN is a foundational algorithm that aims to find the most similar items to a given input. The KNN algorithm calculates the distance between the query vector and each vector in the database based on the chosen distance function, the options are: L2 - Euclidean distance between two vectors, IP - Inner product of two vectors and COSINE - Cosine distance of two vectors.

It then returns the K items with the smallest distances to the query vector. These are the most similar items. Then KNN part of the query searches for the three nearest neighbors. The distance to the query vector is returned as vector_score. The results are sorted by this score.

With the template for the query in place, all query prompts can be executed in a loop by passing the vectorized query prompts over. If the cosine distance or the Euclidean distance are used as the metric, the items with the smallest distance are closer and, therefore, more similar to the query. If the inner product is used as the metric, the items with maximum inner product are more similar to the query. Then, loop over the matched documents and create a list of results that can be converted into a table to visualize the results.

Redis would provide implementation alternatives for the Ingestion operation, along the lines of Section 3.2, as follows (the FLAT alternative will be used as a baseline in Chapter 5):

FLAT – Exhaustive search implementation:

- Uses Redis, with the "FLAT" option (no indexation) and Euclidean distance, to store the full set of vectors. In this option Redis will create the indexes by brute force.

HNSW-NS – HNSW implementation of non-staged ingestion:

– Uses Redis, with the “HNSW” option, which is an implementation of efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs, and Euclidean distance, to store and index the full set of vector.

HNSW-ST – HNSW implementation of the staged ingestion:

– At the end of each stage, uses Redis, with the “HNSW” option and Euclidean distance, to store and index the set of vectors in the cache.

4
A Classified Ad Retrieval Tool

This chapter outlines a proof-of-concept classified ad retrieval tool[1] based on the staged HNSW implementation introduced in Section 3.3 to index the vector streams.

The tool is structured into a main module and three auxiliary modules. The main module is responsible for controlling the task flow between the auxiliary modules and offers a user interface that permits indicating the dataset to be used. The auxiliary modules are a text encoder, an image encoder, and a database. Both the text and the image encoders divide the process of handling data into two steps: the ingestion step and the indexing step. Each step is executed in different servers.

The first section describes the architecture of the tool, followed by its operating process. Then, some use cases and examples are provided to better understand how the tool works.

4.1
Architecture

4.1.1
Technologies Adopted

Multiple resources were used to build the classified ad retrieval tool. How they work together is explained in subsection 4.1.2, after they are detailed in this section.

Python [2] is a high-level, dynamic, interpreted, modular, cross-platform, object-oriented programming language. As it is a relatively user-friendly syntax language, it has gained popularity among professionals in the technology industry who are not specifically programmers, such as engineers, mathematicians, data scientists, researchers and others. One of its biggest attractions is that it has a large number of libraries, both native and third-party, making it very widespread and useful in a wide variety of sectors within web development, and also in areas such as data analysis, machine learning and AI.

The Numpy library [3] provides a large set of library operations and functions that help programmers easily perform numerical calculations. These types

[1]Available at https://github.com/BrunoFMSilva/projeto-final-multimodal-clustering
[2]https://www.python.org
[3]https://numpy.org

of numerical calculations are widely used in tasks such as: Machine Learning Models, Image Processing and Computer Graphics, and Mathematical tasks.

The Pandas library [4] is a Python library for data analysis. It is open source and is free to use. Pandas is built based on two of Python's most famous libraries: matplotlib, for data visualization, and NumPy, for mathematical operations, being a union of these libraries, and allowing many of the matplotlib and NumPy methods to be accessed with less effort. This library is known for its high productivity and performance and its popularity derives from the fact that importing and analyzing data is much more user-friendly.

Docker [5] is a containerization technology for creating and using Linux machines (*containers*). Docker is an extremely lightweight virtual machine-like tool, but it is not actually a virtual machine. It uses (*containers*) which have a different architecture, allowing greater portability and efficiency. The container excludes virtualization and switches the process to Docker. Additionally, *containers* offer greater flexibility for creating, deploying, copying and migrating a *container* from one environment to another.

FastAPI [6] is a modern and high-performance *Web framework* for building APIs with Python. Its main characteristics are:

– Speed, APIs developed with FastAPI have high performance, to the point of being compared with APIs developed with more consolidated technologies.
– Intuitiveness, the *framework* source code was entirely developed using Python's *type hints* feature, this allows you to spend less time debugging the code.
– Lightness, therefore, was entirely designed to be user-friendly, meaning that much less time is spent reading the documentation
– Robustness, in which the developed code is already ready for production, so it is not necessary to make any changes to then put the developed applications live
– Automatic generation of documentation

Redis [7] is a relational database focused on high performance. Its main characteristic is the agility with which it accesses and stores information, largely due to its operating structure. It offers a set of versatile in-memory data structures that enable easy creation of various custom applications.

[4]https://pandas.pydata.org
[5]https://www.docker.com
[6]https://fastapi.tiangolo.com
[7]https://redis.io

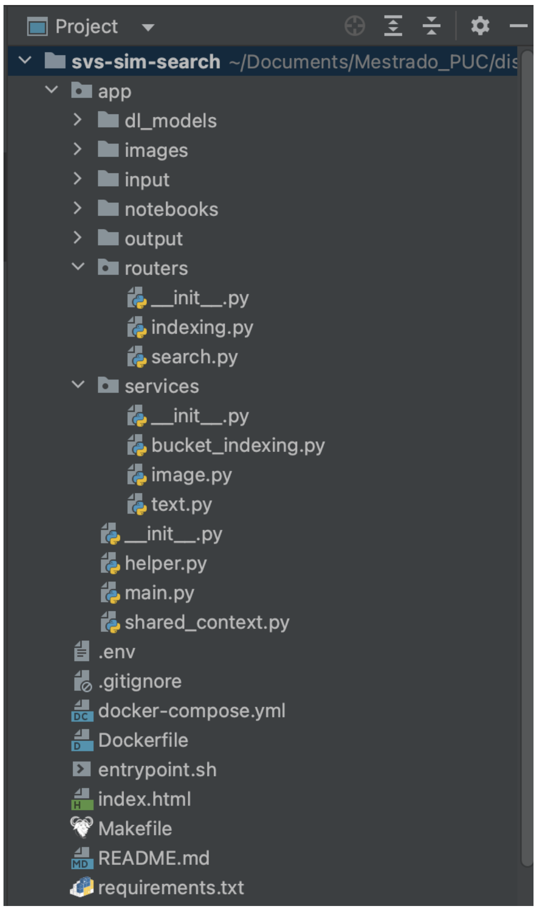
Redis' main use cases include caching, session management, PUB/SUB, and classifications as described in Chapter 3.

4.1.2 Structure

Python was used as the main programming language because of its versatility and friendly syntax. Combined with it, Numpy and Pandas offered the necessary set of mathematical functions and operations in order to manipulate the models used in the embeddings and providing the analysis of the data generated by these models. These mathematical functions are used to calculate the distance between each row in the vectors during the process of indexing.

To create the structure of the application an approach using containers was adopted. These set of containers were built using Docker, due to more organization and less configuration issues related to execution of the application. With these resources set, FastAPI was used to create the graphical interface, since it is a Web Framework to build API's with Python, allowing interactivity between the user and the Tool, managing the text encoder, the image encoder and the database. At last, the database used was Redis which is a relational database that allows different types of customization, like cache, session management, PUB/SUB and ratings. The PUB/SUB requisitions are fundamental in the writing of the index and the properties of the relational database ensure that written data is not lost due to any internal factor, only if the database is destroyed.

Figure 4.1 describes with more details the behavior of the Tool's architecture. The Tool has the folder "app" containing all the information related to the application and the necessary files to execute it, like the containers configuration, the necessary libraries and the environment variables.



```
svs-sim-search ~/Documents/Mestrado_PUC/dis
  app
    dl_models
    images
    input
    notebooks
    output
    routers
      __init__.py
      indexing.py
      search.py
    services
      __init__.py
      bucket_indexing.py
      image.py
      text.py
    __init__.py
    helper.py
    main.py
    shared_context.py
  .env
  .gitignore
  docker-compose.yml
  Dockerfile
  entrypoint.sh
  index.html
  Makefile
  README.md
  requirements.txt
```

Figure 4.1: Project Architecture

Inside the "app" folder many submodules can be observed. Submodule "dl_models" is responsible for storing the models used to process the data sent by the user, converting them to a 768-dimensional vector. Two different models are stored here, one for text encoding and other for image encoding. Their python programs can be seen in Code 1 and Code 2, respectively. The "input" submodule is responsible for storing the data that is entered into the application, containing each of the samples of the dataset. The "output" submodule served as the basis for executing a test that involved pre-processing the input vectors.

The "routers" submodule presents the routes that the application can take when it is already running, allowing it to access the indexing or search part. The "services" submodule displays the services available for the application, the *text* file represents the text encoder, the *image* file represents the image encoder and the *performance_analysis* file aims to evaluate the performance of the indexing and search. Finally, we have three files: *helper*, Code 3, which effectively contains the index creation method. This is called within the text encoder, thus improving readability and reducing code coupling; The *main* file, Code 5, is responsible for creating the *Web* application; *shared_context* is responsible for making the connection to the Redis database, which can be seen in Code 4. Its methods are used throughout the tool, as the database plays a fundamental role in maintaining its structure. It also computes the start of the api, the logging process, load the necessary models in order to encode the input data and the general variables utilized in other modules. The Tool also presents input queries in order to measure its performance and output queries related to the analysis of each metric calculated by the database.

Figure 4.2 describes the UML diagram that represents how this architecture is structured, showing in detail its components and the connections between them. In addition to indicating which components are related to the previously mentioned structures, where the FastAPI block corresponds to the main structure, the text_encoder block corresponds to the text encoder and the Redis block corresponds to the Database.

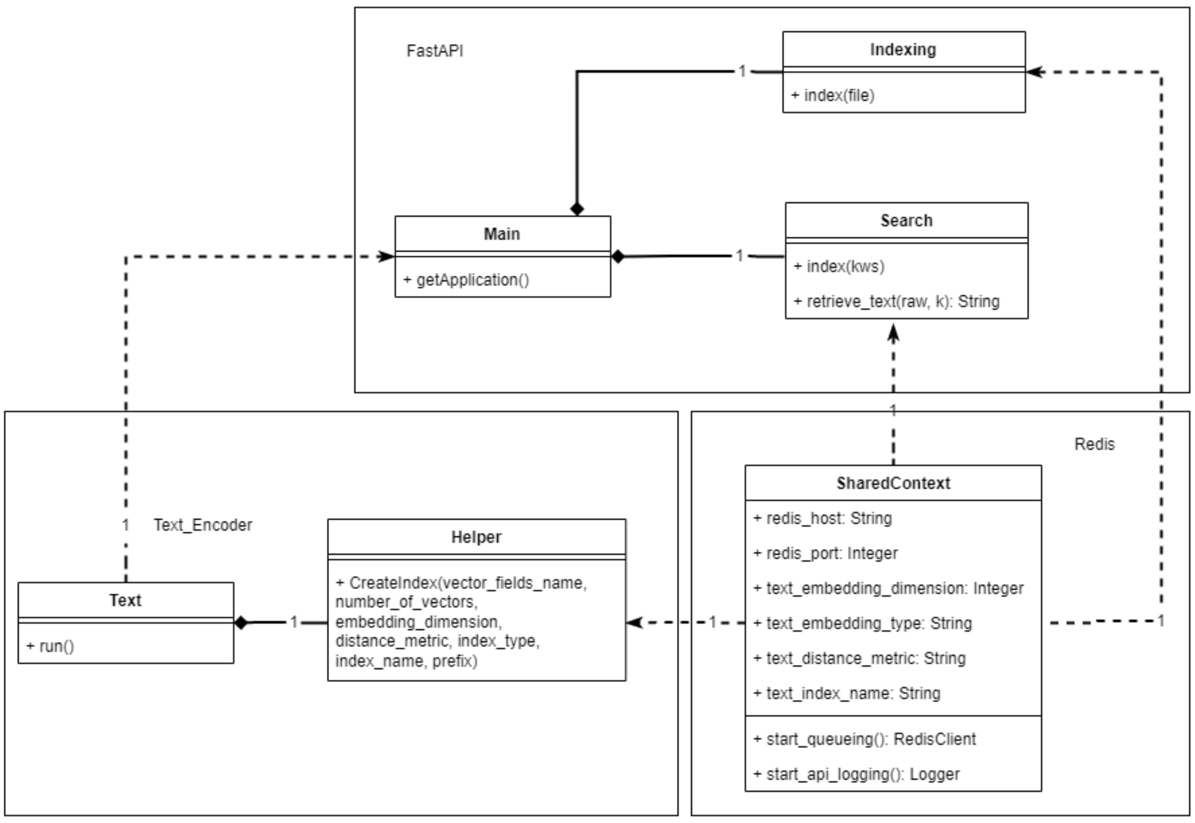

FastAPI
Indexing
+ index(file)
Main
+ getApplication()
Search
+ index(kws)
+ retrieve_text(raw, k): String
Text_Encoder
Text
+ run()
Helper
+ CreateIndex(vector_fields_name, number_of_vectors, embedding_dimension, distance_metric, index_type, index_name, prefix)
Redis
SharedContext
+ redis_host: String
+ redis_port: Integer
+ text_embedding_dimension: Integer
+ text_embedding_type: String
+ text_distance_metric: String
+ text_index_name: String
+ start_queueing(): RedisClient
+ start_api_logging(): Logger
1
1
1
1
1
1
1

Figure 4.2: UML Architecture

4.2 Process

The tool workflow can be divided into two steps: an ingestion step and an indexing step. When the user inserts data that need to be analyzed the application loads the data and starts each step.

The ingestion step runs on the MacBook Pro server equipped with 14 core-GPUs. It uses the Fast API to read the ads from the input files, where each ad has a key, a name, a brief textual description, and an image. The text and image data are then distributed to two queues, corresponding to their data types. When a queue is filled up, the corresponding embeddings are created and exported in a Parquet file. This file type was chosen because of its being strongly typed and having a colunar format which allows a faster reading process and strongly decreases the chance of losing data when reading it. The use of GPUs considerably reduced the time it took to create the embeddings; on average, 36 embeddings were created per second – 20 text embeddings and 16 image embeddings.

The indexing step runs on the PC server with a large amount of main memory to support Redis appropriately. Redis reads the Parquet files and computes the HNSW indices for the embeddings. Therefore, the text and image modules have the same process of operation with some minor differences. The most notable of them is the model used in the ingestion step to create the embbedings of the input data.

On average, it took 6 minutes to generate each index; the image embedding indexing took longer than the text embedding indexing, because the image embeddings were larger than the text embedding.

Following the staged strategy with deferred indexing, the tool buffered 250,000 ads before encoding and storing their text and images.

Finally, to facilitate the experiments, the tool allows the user to test different configurations by varying the embedding dimensions, the type of the indices – "flat" or "HSNW", the distance metrics adopted, and some optimization parameters, such as the construction of the indices in parallel and the amount of memory used.

4.3 Applications of the Tool

The users of this tool are data analysts from companies that deal directly with a large flow of advertisements daily. The analyst, needing to understand the nature and behavior of the data that is in transit in the continuous flow, performs several processes in order to identify the properties of these advertisements. However, due to the enormous volume of ads present in this flow, it becomes extremely exhausting to carry out these analyzes in real time. Therefore, a tool capable of capturing a large number of advertisements and allowing them to be observed at a fixed time becomes extremely useful.

With that said, the tool presented in this dissertation aims to facilitate the data analysis process that analysts need to carry out in their daily lives. This facilitation process starts with searching for specific ads or ad sets. In the tool, the search occurs using keywords related to the scope of the advertisement itself, making it more user-friendly. Furthermore, as it is possible to configure the number of results that the tool returns to the user, the user can search for specific advertisements, checking their existence in the original database or making comparisons with large volumes of data that are more similar to the advertisement. wanted. For example, you can check if any cell phone model is being advertised on the original platform. The tool would return the ads most similar to the one requested, including their description. With these descriptions, it becomes possible to check whether that set of data is coherent or presents any irregularities. Which, in extreme cases, could even mean an indication of fraud.

Another interesting factor in using the tool presented as an analysis mechanism is the fact that duplicate ads are identified in the original database, but this duplication can occur either due to inconsistencies with the database itself or it can identify sellers who are creating multiple advertisements that have the same item and the same description, in order to optimize the sales time of that product.

Finally, this tool can also be used as an ad filter, as when searching for specific keywords it is possible to identify ads that are being placed in the database with inappropriate names or inappropriate descriptions, which allows these ads to be excluded, up to banning those users who are exhibiting this inappropriate behavior, through other processes outside the tool.

4.4 Examples

As an example of text retrieval, suppose the user wants to find the top 3 ads most similar to the ad *"Vendo **Motorola E7**. Vendo Motorola E7, Três **meses de uso** com **nota fiscal**, acompanha capinha e película de vidro"* ("Sell **Motorola E7**. Sell Motorola E7, three **months of use**, with **invoice**, and cover and glass protection cover"). The tool returned:

1. *"**Motorola e7** muito conservado. Vendo Motorola e7 com 4 **meses de uso nota fiscal** e tudo"*
   ("**Motorola e7** in good conditions. Sell Motorola e7 with 4 **months of use invoice** and everything").

2. *"**Motorola E7** semi novo na caixa. Vendo celular Motorola E7 na caixa no valor de 700.00 4 **meses de uso**"*

(“**Motorola E7** almost new in the box. Sell Motorola E7 in the box for 700.00 4 **months of use**”).

3. *“**Motorola E7** só hoje. Vendo esse Motorola E7 valor 500 reais .ele acompanha. Carregado original Fone de ouvido original **Nota fiscal** e caixa. Ele vai fazer 8 **meses de uso**. Motivo da venda *****. ZAP.******** ”*
   (“**Motorola E7** only today. Sell Motorola E7 for 500 reais together with. Original charger Original earphone **invoice** and box. It will be 8 **months old**. Reason *****. ZAP.********”).

In another example of text retrieval, the ad searched was *“**iPhone 13 pro max 128gb** Dourado **lacrado** e com 1 ano de **garantia**.”* (“**iPhone 13 pro max 128gb** Gold **sealed** and with 1 year **warranty**.”). The tool returned:

1. *“**iPhone 13 Pro max 128gb** Aparelho novo **Lacrado** 1 ano de **garantia** Apple Aceito troca por outros iPhones Divido até 12x no cartão.”*
   (“**iPhone 13 Pro max 128gb** New device **Sealed** 1 year Apple **warranty** I accept exchanges for other iPhones Split up to 12x on card”).

2. *“13 pro max **lacrado** 128 GB **iPhone 13 pro max 128GB lacrado** não aceito troca, somente venda !! $6.700 a vista ou 7.437,00 em 12x no cartão.”*
   (“13 pro max **sealed** 128 GB **iPhone 13 pro max 128GB sealed** I do not accept exchanges, sale only !! $6,700 in cash or 7,437.00 in 12 installments on the card.”)

3. *“**iPhone 13 pro max 128gb** Dourado **lacrado** E COM 1 ANO DE **garantia** Fazemos em até 12 vezes A vista tem desconto não entregamos!”*
   (“**iPhone 13 pro max 128gb** Gold **sealed** and with 1 year **warranty** We deliver in up to 12 installments Cash has a discount we do not deliver!”)

As an example of image retrieval, suppose the user wants to find the top 10 ads most similar to an image. Figure 4.3 presents this scenario by showing an image of a hand holding an iPhone on the left and the search results on the right. The tool returned:

– Image 1: the image searched occurs in the first position since it obviously had the highest similarity in the database;

– Images 2 to 4: the dark coloration of the phone was the most relevant fact in the similarity factor;
– Images 5 and 6: the hand holding the phone was considered as one of the most important factors;
– Images 7 to 8: both these factors were considered relevant.

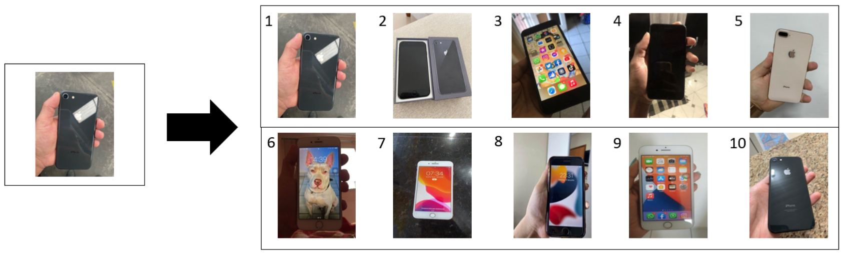


Figure 4.3: Image result set example - Hand Holding iPhone.

In another example of image retrieval, Figure 4.4 presents a scenario where an image of a phone in its home screen is shown. The tool returned multiple phones, half IOS and half Android, with their home screen also being shown. In addition 3 out of 10 images had extra information with it, which consisted in a hand holding the phones and a charger.

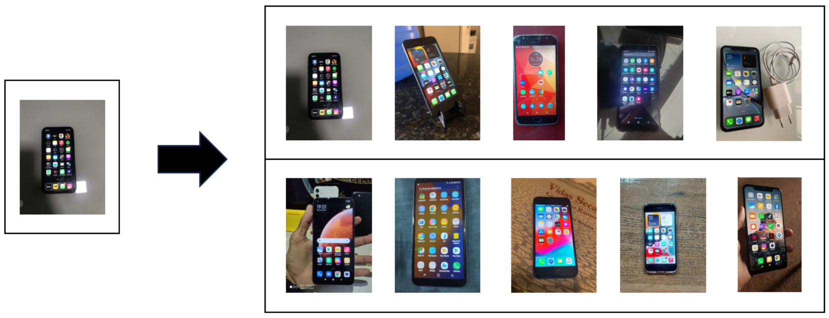


Figure 4.4: Image result set example 2 - Phones with Home Screen being Shown.

Following the last example, Figure 4.5 presents a scenario where an image of a Note 11 is shown. The tool returned in 7 out of 10 images Note 11 phones, one being reflected in the screen of the other phone, and 6 out of 10 were also phones held by hand, being two reflected.

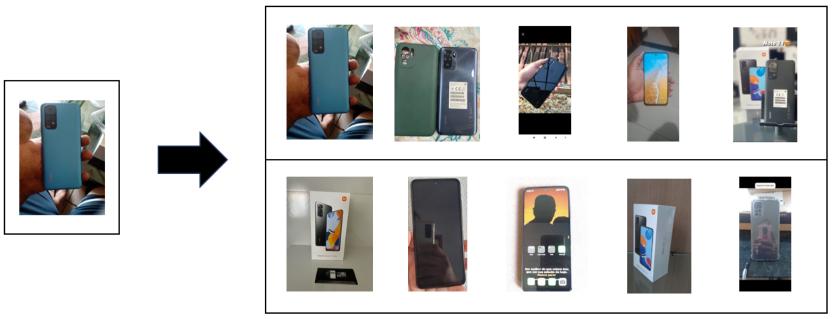


Figure 4.5: Image result set example 3 - Note 11 Phones being Shown by Reflection.

5 Experiments with HNSW

In order to explore the performance of the Classified Ad Retrieval Tool a business dataset was used. This dataset presented a large amount of varied data related to classified ads, which means that the data that composed the main dataset had ads who were similar in a general view but unique in their own perspective. Both of this factors were crucial in the division and analysis presented further in this Section because these factors provided a way for the tool to group the ads and identify its similarities. But in order to limit and analise the data only a portion of the dataset was used, which contained information about different types of eletronic devices, in specific mobile phones.

5.1 Goal

The experiments reported in this section are split into two parts: text analysis considering 50k until 1MM instances and image analysis considering 250k images. These analyses describe experiments to assess the performance of HNSW-ST, the HNSW implementation of the staged ingestion of a stream of vectors with deferred indexing, specifically to:

- *Build cost*: evaluate the cost of building the HNSW index, for various dataset sizes.
- *Query cost*: evaluate the cost of processing a set of queries using HNSW-ST.
- *Search quality*: evaluate the *mean average precision* and *mean average recall* of processing a set of queries using HNSW-ST.

5.2 Datasets

The experiments used data collected from a Brazilian online classified ads company, as follows.

Daily, there is an average of 444k approved ads (about 5/sec) entering the platform. There are three main verticals: `Real Estate`, `Vehicle`, and `Goods`. The experiments target `Goods` ads, focusing on `Electronics > Telephony & Cellphones` (5.89% of approved ads).

Suppose that an ad is a pair $A = (T, I)$, where T is the text description and I is an image associated with the text. For each ad $A = (T, I)$, we created two embeddings, E_T and E_I, such that:

– E_T, the *text embedding*, is a 768-dimensional vector that represents T.
– E_I, the *image embedding*, is a 1,000-dimensional vector that represents I.

To create the text embeddings, we used the transformer "sentence-transformers/ paraphrase-multilingual-mpnet-base-v2"[1] which is based on BERT with the pre-trained weights. This is a sentence paragraph model that maps sentences and paragraphs of 512 chars max length to a 768-dimensional dense vector space. To create the image embeddings, we used the convolutional neural network (CNN) "MobileNet_V2"[2] with the pre-trained weights "IMAGENET1K_V1". One of its main characteristics is the small number of parameters that guarantees high performance. Also, the images are preprocessed with scale adjusts, normalization, and one-hot encoding labels.

We then created 7 datasets:

– *text embeddings datasets:* 6 datasets with the text embeddings of approximately 50k, 100k, 250k, 500k, 750k, and 1MM ads, collected from 2022/06/01 to 2022/07/10. We will denote such datasets as 50k-TE,...,1MM-TE ("TE" stands for "text embeddings").
– *image embeddings dataset:* one dataset, which we will refer to as 250k-IE, with the image embeddings of the same 250k ads.

The experiments to assess index build cost used all 6 text embeddings datasets (Table 5.1) and the image embeddings dataset (Table 5.5) whereas the experiments to assess query cost and search quality used the 1MM-TE and 250k-IE datasets (the other tables in this section).

5.3 Queries

The experiments with the text embeddings datasets adopted 10 text embeddings, $Q_1, ..., Q_{10}$, to play the role of queries, randomly selected from the 1MM-TE dataset. For each query Q_i, the *relevant vectors* were taken as the top-10 text embeddings retrieved by Redis with the "flat" option from the 1MM-TE dataset, which amounts to the 10 vectors closest to Q_i, in Euclidean distance, since Redis, with the "flat" option performs a full dataset scan.

Likewise, the experiments with the image embeddings dataset adopted 10 image embeddings, $P_1, ..., P_{10}$, to play the role of queries, randomly selected from the 250k-IE dataset, and created the set of relevant vectors as before.

[1]https://huggingface.co/sentence-transformers/paraphrase-multilingual-mpnet-base-v2
[2]https://pytorch.org/hub/pytorch_vision_mobilenet_v2/

5.3.1 Hardware and Software Setup

The embeddings were generated on a MacBook Pro with macOS Ventura 13.4, an Apple M1 Pro 8-core processor CPU and 14-core GPU, with 16 GB of RAM and 500 GB of SSD.

Redis was run on a PC server with OS GNU/Linux Ubuntu 16.04.6 LTS, a quad-core processor Intel(R) Core(TM) i7-5820K CPU @ 3.30GHz, with 64 GB of RAM and 1TB of SSD.

HNSW-ST was simulated by dividing each dataset into *batches*. For each batch, Redis was used to create an HNSW index for the vector embeddings of the ad texts and another HNSW index for the vector embeddings of the main ad image. The experiments divided each dataset into 5 batches.

5.3.2 Baselines

The first baseline, called FLAT, was implemented with Redis with the "flat" option (no index) and Euclidean distance and applied to the non-partitioned dataset. The reduction in query processing time, obtained by creating the HNSW index, was then measured against the query processing time of FLAT.

The second baseline was taken as HNSW-NS, as explained in Section 3.3, and applied to the non-partitioned dataset.

5.3.3 Metrics

The experiments adopted as metrics the *total indexing time*, the *total query processing time*, the *mean average recall* (as in Section 3.2), and the *mean average precision*.

5.4 Results

We first present the results for the text embeddings datasets and then those for the image embeddings dataset. To avoid repetition, we postpone a combined discussion of all results to the next subsection.

To evaluate the cost of building the HNSW index, we used all text embeddings datasets and randomly partitioned each one into 5 batches of equal size. Table 5.1 shows the time spent on ingesting the vectors and building the indices in Redis (recall that the FLAT baseline does not build an index) and is organized as follows:

- The lines correspond to the various dataset sizes.
- Several columns correspond to the HNSW-ST simulation:
  - Column “**Batch size**” shows the batch size, which corresponds to the cache size.
  - Columns “**Batch 0**” through “**Batch 4**” show the time Redis took to ingest and build the HNSW index for the vectors in a given batch.
  - Column “**All batches**” shows the sum of the batch times.
- Column “**HNSW-NS**” corresponds to the HNSW-NS baseline and shows the time Redis took to ingest and build the HNSW index for all vectors in a given dataset.

To assess query cost, precision, and recall, we used the 1MM-TE dataset, randomly partitioned into 5 batches of equal size.

Table 5.2 shows the query processing times and is organized as follows (the last column, labeled “**Avg**”, discards the outlier values *min* and *max*):

- Line “**FLAT (k=10)**” corresponds to the FLAT baseline and indicates the time Redis took when adopting no index to retrieve the first k=10 vectors closest to Q_i, using Euclidean distance, from the 1MM-TE dataset.
- Line “**HNSW-NS (k=10)**” corresponds to the HNSW-NS baseline and indicates the time Redis took when adopting the HNSW index to retrieve the first k=10 vectors closest to Q_i, using Euclidean distance, from the 1MM-TE dataset.
- for $i = 0, ..., 4$, line “**HNSW-ST batch i (k=4)**” corresponds to the staged HNSW simulation and indicates the time Redis took when adopting the HNSW index to retrieve the first k=4 vectors closest to Q_i, using Euclidean distance, from the 200,000 vectors in Batch i.

Table 5.3 shows the *mean average precision@k*, for $k = 1, 5, 10$, and is organized as follows:

- Lines labeled “**HNSW-NS**” correspond to the HNSW-NS baseline, that is, to process each query Q_i using Redis with HNSW over the full dataset with 1,000,000 vectors.
- Lines labeled “**HNSW-ST batches**” correspond to the staged HNSW simulation, that is, to processing each query Q_i using Redis with HNSW over each batch with 200,000 vectors, keeping the k=4 first vectors and merging the results.

Table 5.4 shows the *mean average recall@k*, for $k = 1, 5, 10$, and is similarly organized.

Finally, Tables 5.5, 5.6 and 5.7 present the results for the image embeddings dataset. They are organized as the tables for the text embeddings datasets, except for Table 5.5, which shows the indexing times only for the 250k-IE dataset.

Table 5.1: Indexing times (in ms) for various dataset and batch sizes (text-only).

Dataset size	**Batch size**	**Batch 0**	**Batch 1**	**Batch 2**	**Batch 3**	**Batch 4**	**All batches**	**HNSW-NS**
50,000	10,000	7,374	7,883	7,388	7,333	***7,226***	37,204	97,376
100,000	20,000	16,231	15,348	13,251	14,859	13,912	73,601	201,870
250,000	50,000	68,378	63,417	67,307	63,645	103,408	366,155	536,274
500,000	100,000	188,794	252,109	155,201	159,594	105,520	861,218	1,242,132
1,000,000	200,000	244,318	234,918	233,393	232,553	234,731	1,179,913	2,128,172

Table 5.2: Query processing times in ms with the 1MM-TE dataset.

	Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Avg**
FLAT (k=10)	3.41	0.27	0.29	***0.26***	0.27	0.29	0.28	0.27	0.31	0.27	0.28
HNSW-NS (k=10)	0.12	0.05	0.09	***0.05***	0.07	0.09	0.08	0.06	0.14	0.07	0.08
HNSW-ST batch 0 (k=4)	0.08	0.04	0.07	***0.04***	0.05	0.07	0.06	0.05	0.09	0.05	0.06
HNSW-ST batch 1 (k=4)	0.07	0.04	0.06	***0.04***	0.05	0.06	0.05	0.04	0.08	0.05	0.05
HNSW-ST batch 2 (k=4)	0.07	0.04	0.05	***0.03***	0.04	0.05	0.05	0.04	0.08	0.04	0.05
HNSW-ST batch 3 (k=4)	0.07	0.04	0.05	***0.03***	0.04	0.05	0.05	0.04	0.08	0.04	0.05
HNSW-ST batch 4 (k=4)	0.07	0.04	0.05	***0.03***	0.04	0.05	0.05	0.03	0.08	0.04	0.05

Table 5.3: Precision values of the query experiments with the 1MM-TE dataset.

		Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Avg**
HNSW-NS	**precision@1**	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	precision@5	0.20	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.92
	precision@10	0.50	0.90	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.94
HNSW-ST	**precision@1**	1.00	0.00	1.00	0.00	0.00	1.00	0.00	1.00	0.00	1.00	0.50
	precision@5	0.40	0.40	0.60	0.20	0.40	0.40	0.60	0.40	0.60	1.00	0.50
	precision@10	0.50	0.50	0.60	0.40	0.30	0.40	0.60	0.30	0.50	0.80	0.49

Table 5.4: Recall values of the query experiments with the 1MM-TE dataset.

		Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Avg**
HNSW-NS	**recall@1**	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10
	recall@5	0.10	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.46
	recall@10	0.50	0.90	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.94
HNSW-ST	**recall@1**	0.10	0.00	0.10	0.00	0.00	0.10	0.00	0.10	0.00	0.10	0.05
	recall@5	0.20	0.20	0.30	0.10	0.20	0.20	0.30	0.20	0.30	0.50	0.25
	recall@10	0.50	0.50	0.60	0.40	0.30	0.40	0.60	0.30	0.50	0.80	0.49

Table 5.5: Indexing times (in ms) for image dataset and batch sizes (image-only).

Dataset size	**Batch size**	**Batch 0**	**Batch 1**	**Batch 2**	**Batch 3**	**Batch 4**	**All batches**	**HNSW-NS**
250,000	50,000	29,094	29,370	29,522	29,577	29,532	118,209	199,813

Table 5.6: Precision values of the query experiments with the 250k-IE dataset.

		Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Avg
HNSW-NS	precision@1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	precision@5	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	precision@10	1.00	1.00	1.00	1.00	1.00	1.00	0.90	1.00	1.00	1.00	0.99
HNSW-ST	precision@1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	precision@5	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.80	1.00	1.00	0.98
	precision@10	1.00	1.00	0.90	1.00	1.00	1.00	1.00	0.90	1.00	0.90	0.97

Table 5.7: Recall values of the query experiments with the 250k-IE dataset.

		Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Avg
HNSW	recall@1	0,10	0,10	0,10	0,10	0,10	0,10	0,10	0,10	0,10	0,10	0,10
	recall@5	0,50	0,50	0,50	0,50	0,50	0,50	0,50	0,50	0,50	0,50	0,50
	recall@10	1,00	1,00	1,00	1,00	1,00	1,00	0,90	1,00	1,00	1,00	0,99
HNSW batches	recall@1	0,10	0,10	0,10	0,10	0,10	0,10	0,10	0,10	0,10	0,10	0,10
	recall@5	0,50	0,50	0,50	0,50	0,50	0,50	0,50	0,40	0,50	0,50	0,49
	recall@10	1,00	1,00	0,90	1,00	1,00	1,00	1,00	0,90	1,00	0,90	0,97

5.4.1 Discussion

The results obtained for HNSW over the text and image embeddings datasets corroborate the results obtained with the IDFADC implementations.

Recall that the baseline HNSW-NS computed the index for full datasets, whereas HNSW-ST computed a separate index for each batch, which is much faster. Indeed, the sum of the times to ingest and index the vectors for all batches (column labeled "**All batches**" in Tables 5.1 and 5.5) is roughly half of the time to ingest and index the full set of vectors (column labeled "**HNSW-NS**" in Tables 5.1 and 5.5). Furthermore, if we compare the ingestion time of HNSW-NS with the slowest batch, considering a parallel ingestion scenario, we obtain at least a 6-fold speed-up.

Observing Table 5.2, note that the query processing times vary slightly from batch to batch. Also, note that the query processing times using HNSW are about `3.5x` faster than using the "flat" option (no index). The query processing times using the HNSW batches in parallel are between `4.7x` and `6.2x` faster than using the "flat" option. It is important to stress that, since k=4 for the batches, processing the HNSW batches in parallel resulted in $5 \times 4 = 20$ vectors that were sorted by score and filtered to obtain the top 10 vectors.

Finally, as for precision and recall, the results of the experiments with the 1MM-TE dataset (the 1MM text embeddings dataset) show that HNSW-ST, the staged implementation, achieved about half of the performance of the HNSW-NS baseline, on average (column "**Avg**" of Tables 5.3 and 5.4). Again, a possible reason for the decrease in search quality would be that the partitioning and sampling make the data too sparse, which gets accentuated at lower values of R. By contrast, the results of the experiments with the 250k-IE dataset (the 250k image embeddings dataset) show that HNSW-ST achieved roughly the same performance as the HNSW-NS baseline, on average (column "**Avg**" of Tables 5.6 and 5.7).

To summarize, the experiments with real data and HNSW suggest that the staged implementation does not incur significant overhead, and can achieve equivalent search quality. But again the staged implementation scales to vector streams of unbounded length, whereas a non-staged implementation does not.

6
Conclusions

The main contribution of this dissertation was a family of algorithms, called *staged vector stream similarity search – SVS*, to dynamically index a stream of high-dimensional vectors and facilitate similarity search. SVS does not depend on having the full set of vectors available beforehand, but adapts to the vector stream.

SVS provides an elegant solution to the *vector stream similarity search* problem that does not depend on updating the underlying vector index, which is usually expensive, as pointed out in the background and related work section. Indeed, the original contribution of the dissertation stems from the observation that a stream of vectors that become obsolete over time requires an approach different from static vector indexing methods or updating such data structures.

The dissertation discussed two sets of experiments to assess the performance of SVS. The first set of experiments used an IVFADC implementation and the same setup as in (JéGOU; DOUZE; SCHMID, 2011), and the second set adopted an HNSW implementation over real data. These experiments suggested that the SVS implementations do not incur significant overhead and achieve a search quality close to non-staged implementations. Moreover, SVS can support unbounded vector streams.

The dissertation concluded with a brief description of a proof-of-concept implementation of a classified ad retrieval tool, based on Jina and Redis with HNSW. The tool allows retrieving different classified ads, by text or image, in a stream of classified ads vectors provided by the user in the graphical interface. It also permits different configurations by varying the embedding dimensions, the type of the indices, the distance metrics adopted, and some optimization parameters.

As future work, we first plan further experiments with datasets of increasing sizes, of several million vectors, to quantify how IVFADC-ST and IVFADC-NS scale, and with an implementation that would query across the different sets of vectors created at each stage in parallel and would vary the number of vectors retrieved from each stage to achieve the desired recall.

We also plan to conduct further experiments with the proof-of-concept retrieval tool, using much larger datasets collected from the classified ad platform and larger sets of realistic queries. Other types of datasets, unrelated to classified ads, and in different languages are also being considered to expand even further the capabilities of the tool.

A closer look at some query examples also revealed that the ad used to create the first query $Q1$ was duplicated multiple times in the dataset. Then, after a quick validation, it was clear that the seller submitted different ads, which were copies of each other. Thus, the first ten vectors retrieved were from these copies with an Euclidean distance equal to 0. This suggests deduplicating the ads before constructing the test datasets.

7 Bibliography

BENGIO, Y.; COURVILLE, A.; VINCENT, P. Representation learning: A review and new perspectives. **IEEE transactions on pattern analysis and machine intelligence**, IEEE, v. 35, n. 8, p. 1798–1828, 2013.

BEYER, K. et al. When is “nearest neighbor” meaningful? In: SPRINGER. **International conference on database theory**. [S.l.], 1999. p. 217–235.

DATAR, M. et al. Locality-sensitive hashing scheme based on p-stable distributions. In: **Proceedings of the twentieth annual symposium on Computational geometry**. [S.l.: s.n.], 2004. p. 253–262.

FU, C. et al. Fast approximate nearest neighbor search with the navigating spreading-out graph. **Proc. VLDB Endow.**, VLDB Endowment, v. 12, n. 5, p. 461–474, jan 2019. ISSN 2150-8097.

GIONIS, A. et al. Similarity search in high dimensions via hashing. In: **Proc. 25th International Conference on Very Large Data Bases**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999. p. 518–529. ISBN 1558606157.

HAMEED, I. M.; ABDULHUSSAIN, S. H.; MAHMMOD, B. M. Content-based image retrieval: A review of recent trends. **Cogent Engineering**, Cogent OA, v. 8, n. 1, p. 1927469, 2021.

JEGOU, H.; DOUZE, M.; SCHMID, C. Hamming embedding and weak geometric consistency for large scale image search. In: **Computer Vision – ECCV 2008**. [S.l.: s.n.], 2008. p. 304–317.

JOHNSON, J.; DOUZE, M.; JEGOU, H. Billion-scale similarity search with gpus. **IEEE Transactions on Big Data**, IEEE Computer Society, Los Alamitos, CA, USA, v. 7, n. 03, p. 535–547, jul 2021. ISSN 2332-7790.

JéGOU, H.; DOUZE, M.; SCHMID, C. Product quantization for nearest neighbor search. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, v. 33, n. 1, p. 117–128, 2011.

LI, X.; YANG, J.; MA, J. Recent developments of content-based image retrieval (cbir). **Neurocomputing**, v. 452, p. 675–689, 2021. ISSN 0925-2312.

LIU, C. et al. Online optimized product quantization. In: **2020 IEEE International Conference on Data Mining (ICDM)**. [S.l.: s.n.], 2020. p. 362–371.

MALKOV, Y. A.; YASHUNIN, D. A. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. **IEEE Trans. Pattern Anal. Mach. Intell.**, IEEE Computer Society, USA, v. 42, n. 4, p. 824–836, apr 2020. ISSN 0162-8828.

MUJA, M.; LOWE, D. G. Fast approximate nearest neighbors with automatic algorithm configuration. **VISAPP (1)**, v. 2, n. 331-340, p. 2, 2009.

PEREIRA, J. C. et al. On the role of correlation and abstraction in cross-modal multimedia retrieval. **Transactions of Pattern Analysis and Machine Intelligence**, IEEE, v. 36, n. 3, p. 521–535, March 2014. ISSN 0162-8828.

PINHEIRO, J. et al. Indexing high-dimensional vector streams. In: **Proceedings of the 25th International Conference on Enterprise Information Systems**. [S.l.: s.n.], 2023. v. 1.

VASWANI, A. et al. Attention is all you need. In: GUYON, I. et al. (Ed.). **Advances in Neural Information Processing Systems**. [S.l.]: Curran Associates, Inc., 2017. v. 30.

XU, D.; TSANG, I. W.; ZHANG, Y. Online product quantization. **IEEE Transactions on Knowledge and Data Engineering**, v. 30, n. 11, p. 2185–2198, 2018.

YANG, W. et al. Pase: Postgresql ultra-high-dimensional approximate nearest neighbor search extension. In: **Proc. 2020 ACM SIGMOD International Conference on Management of Data**. [S.l.: s.n.], 2020. p. 2241–2253.

YUKAWA, K.; AMAGASA, T. Online optimized product quantization for dynamic database using svd-updating. In: **Database and Expert Systems Applications**. [S.l.: s.n.], 2021. p. 273–284.

ZENG, D.; YU, Y.; OYAMA, K. Deep triplet neural networks with cluster-cca for audio-visual cross-modal retrieval. **ACM Trans. Multimedia Comput. Commun. Appl.**, Association for Computing Machinery, New York, NY, USA, v. 16, n. 3, 2020. ISSN 1551-6857.

A
Appendices

The appendix describes the main codes utilized in the classified Ad Retrieval Tool, showing in depth what each step is responsible for executing within the tool.

A.1
Text Encoder

Code 1: Text Encoder

```python
import sys
from time import sleep
from redis.exceptions import ConnectionError
import app.shared_context as sc
from app.helper import create_index


def run():
    queue_id = int(sys.argv[1]) if len(sys.argv) > 1 else 0
    queue_name = f"{sc.QUEUE_TXT}_{queue_id}"
    sc.api_logger.info(f"consuming from redis streams: {queue_name}")
    last_id_consumed = 0
    sc.api_logger.info("starting loop")
    num_embeddings = 0
    while True:
        try:
            raw_msg = sc.api_redis_cli.xread(count=1, streams={queue_name: last_id_consumed})
        except ConnectionError as exp:
            print(f"...ERROR - {type(exp)} | {exp}...")
            continue
        if not raw_msg:
            sc.api_logger.info(f"empty {queue_name} - skipping...")
            sleep(0.5)
            continue
        try:
            last_id_consumed = raw_msg[0][1][-1][0]
            msg = raw_msg[0][1][-1][1]
            decoded_data = msg.get("data".encode()).decode()
        except (UnicodeDecodeError, AttributeError, ValueError):
            sc.api_logger.info("unicode-decode error detected - skipping")
            sleep(0.5)
            continue
        if decoded_data == sc.START_TOKEN:
            sc.api_logger.info("start token detected")
            sleep(0.5)
            continue
        if decoded_data == sc.END_TOKEN:
            sc.api_logger.info("end token detected")
            sc.api_logger.info(f"{num_embeddings} embeddings inserted")
            sc.api_logger.info("waiting other processes finish")
            sleep(30)
            if queue_id == sc.QUEUE_MAIN:
                sc.api_logger.info("creating index on redis")
                create_index(
                    "idx_txt",
                    sc.TEXT_DISTANCE_METRIC,
                    "embedding",
                    sc.TEXT_EMBEDDING_DIMENSION,
                    "HNSW",
                    "txt::"
                )
                sc.api_logger.info("erasing stream")
                stream_group = sc.api_redis_cli.xread(streams={queue_name: 0})
                for streams in stream_group:
                    stream_name, messages = streams
                    [sc.api_redis_cli.xdel(stream_name, i[0]) for i in messages]
            break
        key, sentence = decoded_data.split(sc.SEPARATOR)
        embeddings = sc.load_txt_model().encode(sentence[:sc.TEXT_MAX_LENGTH])
        sc.api_logger.info(f"key: {key} | embeddings shape: {embeddings.shape}")
        embeddings_bytes = embeddings.astype(sc.TEXT_EMBEDDING_TYPE).tobytes()
        # bucket = int(key) % sc.BUCKETS
        sc.api_redis_cli.hset(
                f"txt::{key}",
                mapping={
                    "embedding": embeddings_bytes,
                    "id": key,
                    "sentence": sentence[:sc.TEXT_MAX_LENGTH]
                }
            )
            num_embeddings += 1


if __name__ == "__main__":
    sc.api_redis_cli = sc.start_queueing(manually=True)
    sc.api_logger = sc.start_encoder_logging()
    run()
```

A.2 Image Encoder

Code 2: Image Encoder

```
import sys
import pandas as pd
import requests as r
import multiprocessing as mp
from redis.exceptions import ConnectionError
from time import sleep, perf_counter
from pathlib import Path
import app.shared_context as sc
from app.helper import create_index, slice_dataframe


input_path = Path(__file__).parent.parent / "input"
images_path = Path(__file__).parent.parent / "images"


def download(idx, sdf):
    dff = pd.read_json(sdf, orient="split")
    print(f"{idx}- df shape: {dff.shape}")

    for i, row in dff.iterrows():
        list_id = row['id']
        image_url = row['image']
        # downloading image
        img_data = r.get(image_url).content
        with open(images_path / f"image_{list_id}.jpg", "wb")
                                                as handler:
```

```
            handler.write(img_data)
        print(f"...{i} - image downloaded | {list_id}...")


def parallel_download(only_missing=False):
    cores = mp.cpu_count()
    df = pd.read_csv(input_path / "electronics_250k.csv")
    print(f"original shape: {df.shape}")
    if only_missing:
        downloaded_images = [int(image_path.name[6:-4]) for
                                                image_path in
                                                images_path.glob("*.
                                                jpg")]
        df = df[~df["id"].isin(downloaded_images)]
        print(f"modified shape: {df.shape}")
    data_frames = slice_dataframe(df, cores)

    procs = []
    start_time = perf_counter()
    for core in range(cores):
        serialized_df = data_frames[core].to_json(orient="
                                                split")
        proc = mp.Process(target=download, args=(core,
                                                serialized_df))
        procs.append(proc)
        proc.start()

    # complete the processes
    for proc in procs:
        proc.join()
    end_time = perf_counter()
    total_time = end_time - start_time
    print(f"Process took {total_time:.4f} seconds")


def run():
    queue_id = int(sys.argv[1]) if len(sys.argv) > 1 else 0
    queue_name = f"{sc.QUEUE_IMG}_{queue_id}"
    sc.api_logger.info(f"consuming from redis streams: {
                                        queue_name}")
    last_id_consumed = 0
    sc.api_logger.info("starting loop")
    num_embeddings = 0
    while True:
        try:
            raw_msg = sc.api_redis_cli.xread(count=1, streams
                                                    ={queue_name:
```

```
                                            last_id_consumed})
        except ConnectionError as exp:
            print(f"ERROR - {type(exp)} | {exp}")
            continue
        if not raw_msg:
            sc.api_logger.info(f"empty {queue_name} -
                                            skipping...")
            sleep(0.5)
            continue
        try:
            last_id_consumed = raw_msg[0][1][-1][0]
            msg = raw_msg[0][1][-1][1]
            decoded_data = msg.get("data".encode()).decode()
        except (UnicodeDecodeError, AttributeError,
                                        ValueError) as exp:
            sc.api_logger.info(f"ERROR - {type(exp)} | {exp}"
                                            )
            sc.api_logger.info(f"unicode-decode error
                                            detected -
                                            skipping")
            sleep(0.5)
            continue
        if decoded_data == sc.START_TOKEN:
            sc.api_logger.info("start token detected")
            sleep(0.5)
            continue
        if decoded_data == sc.END_TOKEN:
            sc.api_logger.info("end token detected")
            sc.api_logger.info(f"{num_embeddings} embeddings
                                            inserted")
            sc.api_logger.info("waiting other processes
                                            finish")
            sleep(30)
            if queue_id == sc.QUEUE_MAIN:
                sc.api_logger.info("creating index on redis")
                create_index(
                    "idx_img",
                    sc.IMG_DISTANCE_METRIC,
                    "embedding",
                    sc.IMG_EMBEDDING_DIMENSION,
                    "HNSW",
                    "img::"
                )
                sc.api_logger.info("erasing stream")
                stream_group = sc.api_redis_cli.xread(streams
                                                ={queue_name:
                                                0})
                for streams in stream_group:
                    stream_name, messages = streams
                    [sc.api_redis_cli.xdel(stream_name, i[0]) for i in messages]
            break
        key, sentence = decoded_data.split(sc.SEPARATOR)
        filename = f"image_{key}.jpg"
        try:
            embeddings = sc.encode_image(img_path=images_path / filename)
        except FileNotFoundError as exp:
            print(f"{exp}")
            continue
        sc.api_logger.info(f"key: {key} | embeddings shape: {embeddings.shape}")
        embeddings_bytes = embeddings.detach()
                            .numpy()
                            .astype(sc.IMG_EMBEDDING_TYPE)
                            .tobytes()
        # bucket = int(key) % sc.BUCKETS
        sc.api_redis_cli.hset(
            f"img::{key}",
            mapping={
                "embedding": embeddings_bytes,
                "id": key
            }
        )
        num_embeddings += 1


if __name__ == '__main__':
    sc.api_redis_cli = sc.start_queueing(manually=True)
    sc.api_logger = sc.start_encoder_logging()
    run()
    # parallel_download()
```

A.3 Helper Library

Code 3: Helper

```
import time
from functools import wraps
import app.shared_context as sc
```

```python
from redis.commands.search.field import VectorField, TextField, NumericField
from redis.commands.search.indexDefinition import IndexDefinition, IndexType


def timeit(func):
    @wraps(func)
    def timeit_wrapper(*args, **kwargs):
        start_time = time.perf_counter()
        result = func(*args, **kwargs)
        end_time = time.perf_counter()
        total_time = end_time - start_time
        print(f'Function {func.__name__}{args} {kwargs} Took {total_time:.4f} seconds')
        return result
    return timeit_wrapper


@timeit
def create_index(vector_field_name,
                    # number_of_vectors,
                    embedding_dimension,
                    distance_metric,
                    index_type="FLAT",
                    index_name="idx_txt",
                    prefix="*"
                    ):
    fields = [
        VectorField(
            vector_field_name,
            index_type,
            {
                "TYPE": "FLOAT32",
                "DIM": embedding_dimension,
                "DISTANCE_METRIC": distance_metric,
                # "INITIAL_CAP": number_of_vectors,
            }
        ),
        TextField("id"),
    ]
    if "txt" in index_name:
        fields.append(TextField("sentence"))
    sc.api_redis_cli.ft(index_name=index_name).create_index(
        fields, definition=IndexDefinition(prefix=[prefix], index_type=IndexType.
                                                          HASH)
    )


def slice_dataframe(dff, qtd):
    num_lines = dff.shape[0]
    lines_per_df = num_lines // qtd
    dfs = []
    ctrl = 0
    for _ in range(qtd - 1):
        dfs.append(dff.iloc[ctrl:ctrl+lines_per_df])
        ctrl += lines_per_df
    dfs.append(dff.iloc[ctrl:])
    return dfs
```

A.4
Shared Context Library

Code 4: Shared Context

```
import sys
import redis
import torch
import logging
import uvicorn
import numpy as np
from PIL import Image, UnidentifiedImageError
from pathlib import Path
from fastapi.logger import logger
from torchvision import transforms
from torchvision.models.mobilenetv2 import
                                    MobileNet_V2_Weights
from sentence_transformers import SentenceTransformer


# TODO: load values from .env file
# constants
REDIS_HOST = "redis"
REDIS_PORT = 6379

QUEUE_TXT = "txt_queue"
QUEUE_IMG = "img_queue"
QUEUE_MAIN = 0  # queue_id responsible to create index
QUEUES_AVAILABLE = 1
SEPARATOR = "|###|"
START_TOKEN = "[STA]"
```

```python
END_TOKEN = "[END]"
BUCKETS = 5

MAX_LOOPS_WITHOUT_DATA = 120  # approximate 1min
TEXT_MAX_LENGTH = 512
TEXT_EMBEDDING_DIMENSION = 768
TEXT_EMBEDDING_FIELD_NAME = "embedding"
TEXT_EMBEDDING_TYPE = np.float32
TEXT_DISTANCE_METRIC = "L2"
TEXT_INDEX_NAME = "idx_txt"

IMG_EMBEDDING_DIMENSION = 1000
IMG_EMBEDDING_FIELD_NAME = "embedding"
IMG_EMBEDDING_TYPE = np.float32
IMG_DISTANCE_METRIC = "L2"
IMG_INDEX_NAME = "idx_img"

API_PORT = 8080
API_HOST = "0.0.0.0"
API_DESCRIPTION = """
## Multimodal Clustering using Product Quantization
"""

# single instances
api_app = None
api_logger = None
api_redis_cli = None
model_txt = None
model_img = None


def start_queueing(manually=False, custom_host="localhost"):
    redis_client = redis.Redis(
        host=REDIS_HOST if not manually else custom_host,
        port=REDIS_PORT,
    )
    return redis_client


def start_api_logging():
    uvicorn_logger = logging.getLogger("uvicorn.access")
    logger.handlers = uvicorn_logger.handlers
    console_formatter = uvicorn.logging.ColourizedFormatter(
        "{message}",
        style="{",
        use_colors=False)
    logger.handlers[0].setFormatter(console_formatter)
    logger.setLevel(uvicorn_logger.level)
    return logger


def start_encoder_logging():
    encoder_logger = logging.getLogger()
    encoder_logger.setLevel(logging.INFO)
    handler = logging.StreamHandler(sys.stdout)
    handler.setLevel(logging.INFO)
    formatter = logging.Formatter('\%(levelname)s:\t\%(message)s')
    handler.setFormatter(formatter)
    encoder_logger.addHandler(handler)
    return encoder_logger


def encode_image(img_path: Path = None,
                 input_image: Image = None):
    global model_img
    # lazy loading
    if not model_img:
        model_img = torch.hub.load(
            "pytorch/vision:v0.10.0",
            "mobilenet_v2",
            weights=MobileNet_V2_Weights.IMAGENET1K_V1
        )
        model_img.eval()
    if not input_image:
        try:
            input_image = Image.open(img_path)
        except UnidentifiedImageError as exp:
            raise FileNotFoundError(f"...ERROR - {type(exp)} | {exp}...")
    preprocess = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
    ])
    input_tensor = preprocess(input_image)
    input_batch = input_tensor.unsqueeze(0)  # create a mini-batch as expected by the model

    if torch.cuda.is_available():
```

```python
        input_batch = input_batch.to("cuda")
        model_img.to("cuda")

    with torch.no_grad():
        output = model_img(input_batch)
    # Tensor of shape 1000, with confidence scores over Imagenet's 1000 classes
    # print(output[0])
    # The output has unnormalized scores. To get probabilities, you can run a softmax on it.
    embeddings = torch.nn.functional.softmax(output[0], dim=0)
    return embeddings


def load_txt_model():
    global model_txt
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    # lazy loading
    if not model_txt:
        model_txt = SentenceTransformer(
            model_name_or_path="sentence-transformers/paraphrase-multilingual-mpnet-base-v2",
            cache_folder=str(Path(__file__).parent / "dl_models"),
            device=str(device),
        )
    return model_txt
```

A.5 Main Code

Code 5: Main

```python
import uvicorn
import app.shared_context as sc
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from app.routers import indexing, search


def get_application() -> FastAPI:
    app = FastAPI(
        title="Multimodal Clustering",
        description=sc.API_DESCRIPTION,
        version="0.0.1",
    )
    app.include_router(indexing.router)
    app.include_router(search.router)

    origins = [
        "http://localhost",
        "http://localhost:8080",
        "http://localhost:63342",
    ]

    app.add_middleware(
        CORSMiddleware,
        allow_origins=origins,
        allow_credentials=True,
        allow_methods=["*"],
        allow_headers=["*"],
    )

    @app.on_event("startup")
    def startup_event():
        sc.api_redis_cli = sc.start_queueing()
        sc.api_logger = sc.start_api_logging()

    @app.get("/healthcheck", include_in_schema=False)
    def healthcheck():
        return {"status": "ok"}

    return app


sc.api_app = get_application()


if __name__ == '__main__':
    uvicorn.run(sc.api_app, host=sc.API_HOST, port=sc.
                                        API_PORT)
```