

Staged Vector Stream Similarity Search Methods

João P.V. Pinheiro¹[0000-0002-0909-4432], Lucas R. Borges¹, Bruno F. Martins da Silva¹[0009-0002-5076-6808], Luiz A.P. Paes Leme²[0000-0001-6014-7256], and Marco A. Casanova¹[0000-0003-0765-9636]

¹ Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro RJ, Brazil

² Universidade Federal Fluminense, Niterói RJ, Brazil

{jpinheiro, lborges, bsilva}@inf.puc-rio.br, lapaesleme@ic.uff.br,
casanova@inf.puc-rio.br

Abstract. This article describes the staged vector stream similarity search methods, or briefly SVS, designed to index and search vector streams by similarity over a time interval. SVS continuously adapts to the vector stream as the vectors are received and do not depend on costly updates on an index structure. The article presents experiments to investigate the performance of two implementations of SVS, one based on product quantization and another based on Hierarchical Navigable Small World graphs. Finally, the article describes a proof-of-concept implementation of a classified ad retrieval tool that uses staged HNSW on real data collected from an online classified ads company.

Keywords: High-dimensional Vector Streams, Approximate Nearest Neighbor Search, Product Quantization, Hierarchical Navigable Small World Graphs, Classified Ad.

1 Introduction

This article 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 complete 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.

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, and buyers can search for products. The tool would combine text and content-based image retrieval [8, 12], 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

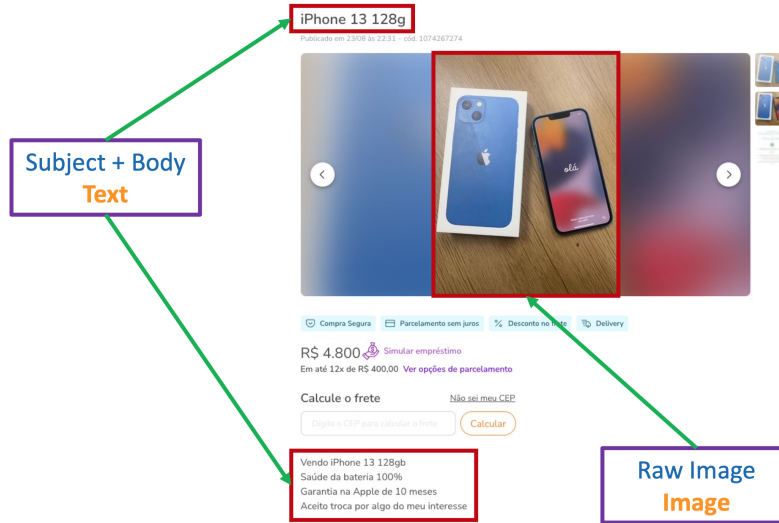


Fig. 1. Example of a classified ad [17].

high-dimensional vector, as in cross-modal retrieval techniques [3, 22]. In either case, the challenge lies in implementing approximated nearest neighbor search over high-dimensional vectors [10, 11, 20]. 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 withdrew 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 article 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 article presents two implementations of SVS: one is based on IVFADC - “Inverted File with Asymmetric Distance Computation” [11], and is called *staged IVFADC*; and another is based on HNSW - “Hierarchical Navigable Small World” graphs [14], as implemented in Redis³, and is called *staged HNSW*. IVFADC and HNSW were chosen since they are well-known approximated vector similarity search methods.

Next, the article describes two sets of experiments to assess the implementations. The experiments with staged IVFADC adopt the database and query

³ <https://redis.io>

descriptors from the INRIA Holidays images [9], 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 article 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 [17]. This article clarifies the description of SVS, provides additional details about staged IVFADC and related experiments, expands the experiments with staged HNSW, and details the proof-of-concept implementation.

The rest of this article is organized as follows. Section 2 covers related work. Section 3 introduces SVS. Section 4 describes the staged IVFADC and staged HNSW implementations. Section 5 contains sets of experiments with the implementations. Section 6 outlines the proof-of-concept implementation. Finally, Section 7 presents the conclusions.

2 Background and Related Work

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

2.1 Neural Network Architectures

Transformers. The transformer is a model architecture relying 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 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 and 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. Like the encoder, residual connections are employed around each sub-layer, followed by layer normalization. Then, the self-attention sub-layer in the decoder stack is modified to prevent positions from attending to subsequent positions. This

masking, combined with the 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 [18].

Convolutional Neural Networks. Convolutional Neural Networks (CNNs) are feed-forward networks in that information flow takes place in one direction only, from their inputs 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. CNN architectures consist of convolutional and pooling (or subsampling) layers grouped into modules. Either one or more fully connected layers, as in a standard feed-forward neural network, follow these modules. Modules are often stacked on top of each other to form a deep model.

2.2 Vector Indexing Methods, Libraries, and Search Engines

Indexing Methods. Similarity search on large scale, high dimensional datasets is an essential feature of several Deep Learning applications [1]. 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 [2] and traditional indexing strategies do not fare much better [11]. Several algorithms [4, 7, 15] 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 [11] 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 [11] 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 [10].

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. [14] proposed the *Hierarchical Navigable Small World – HNSW* index for the approximate k -nearest neighbor search based on navigable small-world graphs with controllable hierarchy. HNSW incrementally builds a multi-layer structure consisting of a hierarchical set of proximity graphs (layers) for nested subsets of the stored elements. HNSW starts a search by randomly selecting an entry node from the top layer, and goes through all neighbor nodes of the entry node. It repeatedly explores the neighbors of new candidate nodes, and maintains a list of k items, ordered based on the distance to the target. HNSW performs very well even on a large dataset, and can obtain a higher speedup than a quantization-based algorithm. However, HNSW spends a relatively long time building neighbor graphs. Graph storage is another bottleneck when the dataset is too large [6].

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 1.

FAISS⁴ 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. ScaNN⁵ is a similar library developed at Google. NGT⁶ - 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. [20] 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. Milvus⁷ is another example of a vector database offering similarity search. It supports, among others, IVFFlat and HNSW. Weaviate⁸ 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. Qdrant⁹ and Elastic¹⁰ are other examples of vector search engines.

⁴ <https://github.com/facebookresearch/faiss/wiki/>

⁵ <https://github.com/google-research/google-research/tree/master/scann>

⁶ <https://morioh.com/p/8c38367453ae>

⁷ <https://milvus.io/docs/index.md>

⁸ <https://weaviate.io>

⁹ <https://qdrant.tech>

¹⁰ <https://www.elastic.co/what-is/vector-search>

Summary. Table 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-Benchmarks¹¹, a benchmarking environment for approximate nearest neighbor search algorithms.

Section 4 describes staged implementations based on IVFADC and HNSW, while Section 5 assesses the overhead of the staged implementations against non-staged, equivalent baselines.

Table 1. A comparison of vector indexing libraries and search engines [17].

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.

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

Xu et al. [19] 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. [13] also proposed an online, optimized product quantization model to dynamically update the codebooks and the rotation matrix.

Yukawa and Amagasa [21] 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

¹¹ <http://ann-benchmarks.com/index.html>

uses the stored indexes to process approximated nearest neighbor search over the high-dimensional vectors.

3 SVS – The Family of 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 2 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 2 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. [17] 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.

Table 2. 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

In either case, I is persisted in secondary storage when the stage ends, and reinitialized for the next stage. This article 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 cover. 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:    $t_b \leftarrow \text{CLOCK}$ 
4:   repeat
5:     READ( $V; v$ ) ▷ read  $v$  from stream  $V$ 
6:      $t \leftarrow \text{CLOCK}$ 
7:     ADDCACHE( $(v, t), C$ )
8:      $e \leftarrow (\text{CLOCK} - t_b)$  ▷ cache elapsed time
9:     if  $C$  is full or  $e > \textit{timeout}$  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:       $T \leftarrow (t_b, \text{CLOCK})$  ▷ time interval
16:      STORE( $(I, T)$ )
17:       $C \leftarrow \emptyset$ 
18:       $t_b \leftarrow \text{CLOCK}$ 
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

```

4 Implementations of the Ingestion Operation

4.1 Implementations based on IVFADC

IVFADC, outlined in Section 2.2, 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. [11] 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 [20] 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.

4.2 Implementations based on HNSW

Redis¹² would provide implementation alternatives for the INGESTION operation, along the lines of Section 4.1, as follows (the FLAT alternative will be used as a baseline in Section 5.2):

FLAT – Exhaustive search implementation:

- Uses Redis, with the “FLAT” option (no indexation) and Euclidean distance, to store the full set of vectors.

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

- Uses Redis, with the “HNSW” option 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.

5 Experiments

5.1 Experiments with Product Quantization

Goal. The experiments reported in this section assess the performance of IVFADC-ST, the IVFADC implementation of the staged ingestion of a stream of vectors, specifically to:

- *Build cost*: evaluate the cost of IVFADC-ST, for various cache sizes.
- *Query cost*: compare the cost of processing a set of queries when IVFADC-ST and IVFADC-NS (the baseline) are adopted.
- *Search quality*: compare the *mean average recall@R* of processing a set of queries when IVFADC-ST and IVFADC-NS are adopted.

Dataset and queries. The experiments used:

- Dataset: a random partition of the 1 million INRIA Holidays images¹³ into 10 *batches* B_1, \dots, B_{10} .
- Query descriptors: from the INRIA Holidays images.

¹² <https://redis.io>

¹³ <https://lear.inrialpes.fr/~jegou/data.php#holidays>

HW and SW setup. All experiments used a PC with an Intel core i5-9600k CPU @ 3.7GHz processor and 16 GB RAM (12GB for the VM), running Ubuntu 20.04 on WSL2 VM and Python 3.10.

The codebooks and the vectors were stored in main memory, and the Euclidean distance was adopted, as in the original experiments reported in [11].

To simulate the use of a cache that holds 100,000 images, the dataset was partitioned into 10 sets. That is, IVFADC simply processes each batch B_i and trains a codebook with a sample of B_i . Note that this simple strategy greatly facilitates the experiments since it just simulates IVFADC-ST using a standard implementation of IVFADC.

Baseline. The baseline was taken as IVFADC-NS applied to the non-partitioned dataset, trained with a sample of the dataset (rather than a separate training set as in [11]).

Metrics. As search quality metric, the experiments adopted the *mean average recall@R*, as in [11]. In general, given a query Q with a set r_Q of relevant vectors for Q , *recall@R* is the proportion of vectors in r_Q which are ranked in the first R positions. In this particular case, the set of relevant vectors for Q is a group of vectors closest to Q , as specified in [9]¹⁴.

Results. The results showed that:

1. IVFADC-ST and the baseline IVFADC-NS had equivalent total build cost, both in terms of training the codebooks and indexing the data.
2. IVFADC-ST incurred a significant increase of around 40% in query cost when querying across all 10 batches (the query cost was not detailed in a separate table since they were uniformly 40% higher).
3. As observed in Table 3, when compared with the baseline IVFADC-NS, IVFADC-ST incurred a slight reduction of *recall@R*, for lower values of R , but had similar *recall@R*, otherwise.

Table 3. Recall values of the staged product quantization experiments [17].

	recall@1	recall@5	recall@10	recall@20	recall@50	recall@100
IVFADC-NS (Baseline)	0.269	0.534	0.646	0.743	0.826	0.859
IVFADC-ST	0.251	0.504	0.618	0.725	0.823	0.865

¹⁴ See also <http://lear.inrialpes.fr/people/jegou/data.php>

Discussion. The results deserve a few comments. First, note that, if a new batch is considered, the baseline IDFADC-NS would have to recompute the codebook from a sample of the set of all vectors. On the other hand, IVFADC-ST would only have the cost associated with processing the new batch. Further experiments with datasets of increasing sizes, with several million vectors, would be required to quantify how IVFADC-ST and IVFADC-NS scale.

Second, the increase in query cost comes from the need to query across the different codebooks, calculating the residual distance to each different product quantizer centroid, and merging the individual result lists. Evidently, the query cost depends on how many different batches span the relevant interval. Thus, narrow intervals could significantly lower the query cost by limiting the number of batches that must be searched. Query cost could also be reduced by adopting an implementation that would query across the different sets of vectors created at each stage in parallel. Further experiments would again be required to quantify how much reduction could be obtained with such implementation.

Third, 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 . Once again, further experiments would be required to vary the number of vectors retrieved from each stage to achieve the desired recall, after merging and ranking the partial lists of vectors retrieved.

To summarize, these early experiments suggest that IVFADC-ST does not incur significant overhead, can achieve equivalent search quality, and has a query cost that depends on the search interval. But IVFADC-ST scales to vector streams of unbounded length, whereas IDFADC-NS, the non-staged implementation, does not.

5.2 Experiments with HNSW

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@R* and *mean average recall@R* of processing a set of queries using HNSW-ST.

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”¹⁵ 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”¹⁶ 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 4) and the image embeddings dataset (Table 8) whereas the experiments to assess query cost and search quality used the 1MM-TE and 250k-IE datasets (the other tables in this section).

Queries. The experiments with the text embeddings datasets adopted 10 text embeddings, Q_1, \dots, 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, \dots, P_{10} , to play the role of queries, randomly selected from the 250k-IE dataset, and created the set of relevant vectors as before.

HW and SW 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.

¹⁵ <https://huggingface.co/sentence-transformers/paraphrase-multilingual-mpnet-base-v2>

¹⁶ https://pytorch.org/hub/pytorch_vision_mobilenet_v2/

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.

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 4.2, and applied to the non-partitioned dataset.

Metrics. The experiments adopted as metrics the *total indexing time*, the *total query processing time*, the *mean average recall@R* (as in Section 4.1), and the *mean average precision@R*.

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 4 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 simulates 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 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 $i = 0, \dots, 4$, 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 6 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 7 shows the *mean average recall@k*, for $k = 1, 5, 10$, and is similarly organized.

Finally, Tables 8, 9 and 10 present the results for the image embeddings dataset. They are organized as the tables for the text embeddings datasets, except for Table 8, which shows the indexing times only for the 250k-IE dataset.

Table 4. Indexing times (in ms) for various dataset and batch sizes (text-only) [17].

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

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 4 and 8) is roughly half of the time

Table 5. Query processing times in ms with the 1MM-TE dataset [17].

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Avg
FLAT (k=10)	3.4132	0.2694	0.2859	<i>0.2629</i>	0.2718	0.2855	0.2800	0.2687	0.3117	0.2714	0.2806
HNSW-NS (k=10)	0.1241	0.0523	0.0935	<i>0.0463</i>	0.0697	0.0871	0.0793	0.0623	0.1401	0.0684	0.0796
HNSW-ST batch 0 (k=4)	0.0831	0.0444	0.0676	<i>0.0394</i>	0.0534	0.0664	0.0600	0.0487	0.0925	0.0526	0.0595
HNSW-ST batch 1 (k=4)	0.0689	0.0407	0.0577	<i>0.0358</i>	0.0462	0.0571	0.0519	0.0421	0.0776	0.0457	0.0513
HNSW-ST batch 2 (k=4)	0.0660	0.0385	0.0533	<i>0.0335</i>	0.0412	0.0522	0.0470	0.0382	0.0764	0.0415	0.0472
HNSW-ST batch 3 (k=4)	0.0660	0.0368	0.0521	<i>0.0316</i>	0.0382	0.0506	0.0457	0.0356	0.0765	0.0385	0.0454
HNSW-ST batch 4 (k=4)	0.0660	0.0358	0.0519	<i>0.0311</i>	0.0380	0.0528	0.0454	0.0341	0.0765	0.0373	0.0452

Table 6. Precision values of the query experiments with the 1MM-TE dataset [17].

		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 7. Recall values of the query experiments with the 1MM-TE dataset [17].

		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 8. 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 9. 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 10. 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

to ingest and index the full set of vectors (column labeled “HNSW-NS” in Tables 4 and 8). 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, 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 6 and 7). 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 9 and 10).

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 A Classified Ad Retrieval Tool

This section outlines a proof-of-concept classified ad retrieval tool¹⁷ based on the staged HNSW implementation introduced in Section 4.2 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, among other features. 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 executes on a different server.

The ingestion step runs on the MacBook Pro server equipped with 14 core-GPUs. It uses the Fast API [5] 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 [16]. This file type was chosen because of its

¹⁷ Available at <https://github.com/BrunoFMSilva/projeto-final-multimodal-clustering>

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, as briefly explained in the **datasets** subsection of Section 5.2. 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.

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.*****”).

As an example of image retrieval, suppose the user wants to find the top 10 ads most similar to an image. Figure 2 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.

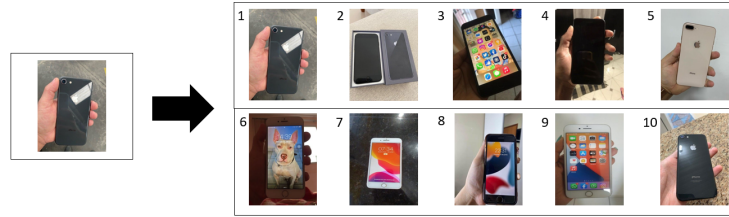


Fig. 2. Image result set example.

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 “HNSW”, the distance metrics adopted, and some optimization parameters, such as the construction of the indices in parallel and the amount of memory used.

7 Conclusions

This article described the staged vector stream similarity search methods (SVS) designed to index and search vector streams by similarity over a time interval. SVS continuously adapts to the vector stream as the vectors are received and do not depend on costly updates on an index structure. The article presented experiments to investigate the performance of two implementations of SVS, one based on product quantization, and another based on Hierarchical Navigable Small World graphs. The experiments indicate that the SVS implementations achieved a search quality close to non-staged implementations and yet they can support unbounded vector streams. Finally, the article described a proof-of-concept implementation of a classified ad retrieval tool that uses Redis with HNSW on real data collected from an online classified ads company.

As future work, we first plan further experiments with datasets of increasing sizes, of several million vectors, to quantify how the implementations scale. In particular, we plan to run the proof-of-concept retrieval tool on much larger datasets collected from the classified ad platform and larger sets of realistic queries. A closer look at query examples also revealed that some of the ads were 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. This suggests deduplicating the ads before constructing the test datasets.

Acknowledgements

This work was partly funded by FAPERJ under grants E-26/200.834/2021, by CAPES under grant 88881.134081/2016-01 and 88882.164913/2010-01, and by CNPq under grant 305.587/2021-8.

References

1. Bengio, Y., Courville, A., Vincent, P.: Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence* **35**(8), 1798–1828 (2013). <https://doi.org/10.1109/TPAMI.2013.50>
2. Beyer, K., Goldstein, J., Ramakrishnan, R., Shaft, U.: When is “nearest neighbor” meaningful? In: *International conference on database theory*. pp. 217–235. Springer (1999). https://doi.org/10.1007/3-540-49257-7_15
3. Costa Pereira, J., Coviello, E., Doyle, G., Rasiwasia, N., Lanckriet, G., Levy, R., Vasconcelos, N.: On the role of correlation and abstraction in cross-modal multimedia retrieval. *Transactions of Pattern Analysis and Machine Intelligence* **36**(3), 521–535 (March 2014). <https://doi.org/10.1109/TPAMI.2013.142>
4. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-sensitive hashing scheme based on p-stable distributions. In: *Proceedings of the twentieth annual symposium on Computational geometry*. pp. 253–262 (2004). <https://doi.org/10.1145/997817.997857>
5. Fast Api, <https://fastapi.tiangolo.com>
6. Fu, C., Xiang, C., Wang, C., Cai, D.: Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proc. VLDB Endow.* **12**(5), 461–474 (jan 2019). <https://doi.org/10.14778/3303753.3303754>
7. Gionis, A., Indyk, P., Motwani, R., et al.: Similarity search in high dimensions via hashing. In: *Proc. 25th International Conference on Very Large Data Bases*. p. 518–529. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1999)
8. Hameed, I.M., Abdulhussain, S.H., Mahmmud, B.M.: Content-based image retrieval: A review of recent trends. *Cogent Engineering* **8**(1), 1927469 (2021). <https://doi.org/10.1080/23311916.2021.1927469>
9. Jegou, H., Douze, M., Schmid, C.: Hamming embedding and weak geometric consistency for large scale image search. In: *Computer Vision – ECCV 2008*. pp. 304–317 (2008). https://doi.org/10.1007/978-3-540-88682-2_24
10. Johnson, J., Douze, M., Jegou, H.: Billion-scale similarity search with gpus. *IEEE Transactions on Big Data* **7**(03), 535–547 (jul 2021). <https://doi.org/10.1109/TBDATA.2019.2921572>
11. Jégou, H., Douze, M., Schmid, C.: Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **33**(1), 117–128 (2011). <https://doi.org/10.1109/TPAMI.2010.57>
12. Li, X., Yang, J., Ma, J.: Recent developments of content-based image retrieval (cbir). *Neurocomputing* **452**, 675–689 (2021). <https://doi.org/10.1016/j.neucom.2020.07.139>
13. Liu, C., Lian, D., Nie, M., Hu, X.: Online optimized product quantization. In: *2020 IEEE International Conference on Data Mining (ICDM)*. pp. 362–371 (2020). <https://doi.org/10.1109/ICDM50108.2020.00045>
14. 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.* **42**(4), 824–836 (apr 2020). <https://doi.org/10.1109/TPAMI.2018.2889473>
15. Muja, M., Lowe, D.G.: Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP (1)* **2**(331-340), 2 (2009). <https://doi.org/10.5220/0001787803310340>
16. Apache Parquet, <https://parquet.apache.org/docs/>

17. Pinheiro, J., Borges, L., Silva, B., Leme, L., Casanova, M.: Indexing high-dimensional vector streams. In: Proceedings of the 25th International Conference on Enterprise Information Systems. vol. 1 (2023). <https://doi.org/10.5220/0011758900003467>
18. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L.u., Polosukhin, I.: Attention is all you need. In: Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R. (eds.) Advances in Neural Information Processing Systems. vol. 30. Curran Associates, Inc. (2017)
19. Xu, D., Tsang, I.W., Zhang, Y.: Online product quantization. *IEEE Transactions on Knowledge and Data Engineering* **30**(11), 2185–2198 (2018). <https://doi.org/10.1109/TKDE.2018.2817526>
20. Yang, W., Li, T., Fang, G., Wei, H.: Pase: Postgresql ultra-high-dimensional approximate nearest neighbor search extension. In: Proc. 2020 ACM SIGMOD International Conference on Management of Data. p. 2241–2253 (2020). <https://doi.org/10.1145/3318464.3386131>
21. Yukawa, K., Amagasa, T.: Online optimized product quantization for dynamic database using svd-updating. In: Database and Expert Systems Applications. pp. 273–284 (2021). https://doi.org/10.1007/978-3-030-86472-9_25
22. 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.* **16**(3) (2020). <https://doi.org/10.1145/3387164>