Indexing High-Dimensional Vector Streams

João Pedro V. Pinheiro[1] [a], Lucas Ribeiro Borges[1],
Bruno Francisco Martins da Silva[1], Luiz André P. Paes Leme[2] [b], and Marco Antonio Casanova[1] [c]

[1]*Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro RJ, Brazil*
jpinheiro@inf.puc-rio.br, lucasrborges94@gmail.com, bsilva@inf.puc-rio.br, casanova@inf.puc-rio.br

[2]*Universidade Federal Fluminense, Niterói RJ, Brazil*
lapaesleme@ic.uff.br

Abstract: This paper addresses 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 paper first introduces a family of methods, called *staged vector stream similarity search methods*, or briefly *SVS methods*, to help solve this problem. SVS methods are continuous in the sense that they do not depend on having the full set of vectors available beforehand, but adapt to the vector stream. The paper then presents experiments to assess the performance of two SVS methods, one based on product quantization, called staged IVFADC, and another based on Hierarchical Navigable Small World graphs, called staged HNSW. The experiments with staged IVFADC use well-known image datasets, while those with staged HNSW use real data. The paper concludes with a brief description of a proof-of-concept implementation of a classified ad retrieval tool that uses staged HNSW.

1 Introduction

A *classified ad* is a textual description of a product, with a few images of the product, and the price, sale conditions, and the required seller data - Figure 1. An *online trading platform*, or simply a *trading platform*, is used in this paper in the specific sense of a Web application where sellers can post classified ads and buyers can search for products and close transactions. The transaction volume a trading platform must support can be significant. Indeed, a popular online product trading platform typically processes thousands of classified ads per day.

The motivation for this paper lies in the challenge of creating a *classified ad retrieval tool* that receives a classified ad and returns a ranked list of similar ads. Similarity, in this case, would be computed with respect to the textual description, the set of images, price, etc. of the classified ads. The paper considers the scenario where new ads are continuously included in the platform, creating a stream of classified ads.

The construction of a classified ad retrieval tool, in this scenario, must face two major difficulties. First, the tool would have to combine text and content-based image retrieval, since product descriptions contain text and images. Albeit there are several well-tested text retrieval tools, retrieving images by content similarity is still challenging, especially when the number of images is high. State-of-the-art content-based image retrieval strategies (Hameed et al., 2021; Li et al., 2021) assume that images are represented by high-dimensional vectors, created using some Deep Learning technique. Alternatively, the tool might transform both the text and the images (or in fact any other media as well) of an ad into a single high-dimensional vector, as in cross-modal retrieval techniques (Costa Pereira et al., 2014; Zeng et al., 2020). The challenge then becomes how to efficiently search a (large set) of high-dimensional vectors or, more precisely, how to implement approximated nearest neighbor search over high-dimensional vectors (Jégou et al., 2011; Johnson et al., 2021; Yang et al., 2020).

The second difficulty 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

---

[a] [iD] https://orcid.org/0000-0002-0909-4432
[b] [iD] https://orcid.org/0000-0001-6014-7256
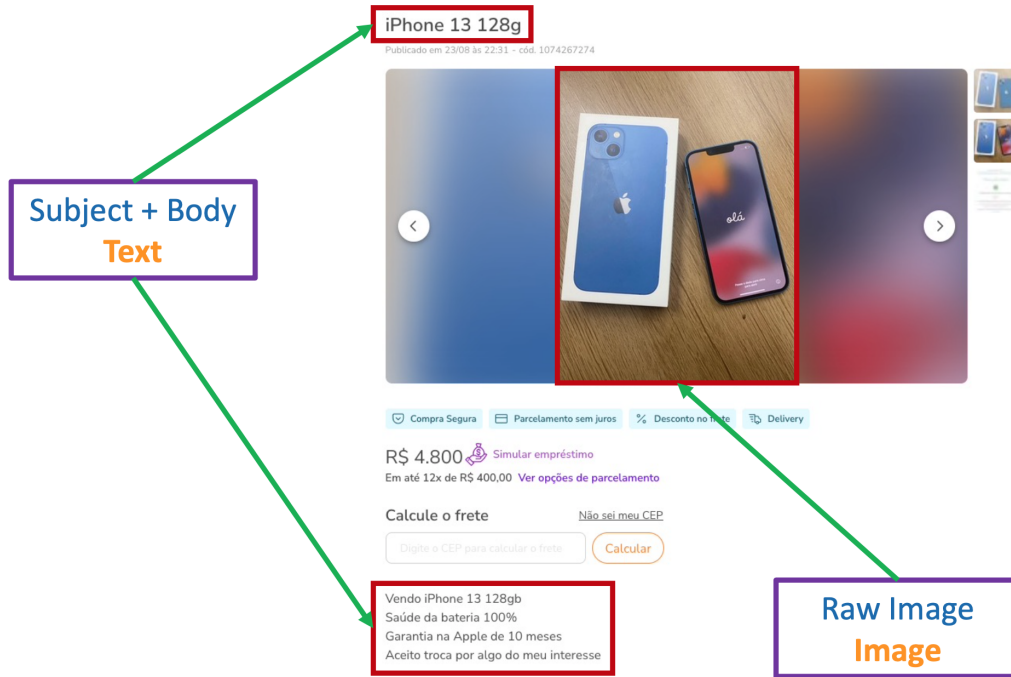[c] [iD] https://orcid.org/0000-0003-0765-9636

Figure 1: Example of a classified ad.

short-lived, either because the product was actually sold, or because the seller withdraw the ad, or simply because the ad became obsolete for some reason. In fact, one may model this scenario as a *classified ad stream*, where the retrieval process occurs over the stream, perhaps limited to some point in the past, as a sliding time window. This characteristic of trading platforms therefore requires that the retrieval strategies support a dynamic scenario.

From a high-level point of view, this scenario would require solving 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 main contribution of the paper is a family of methods, called *staged vector stream similarity search methods*, or briefly *SVS methods*, to help solve this problem. An SVS method uses 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, an SVS method 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.

The paper discusses experiments that assess the performance of implementations of two SVS methods: one is based on IVFADC - "Inverted File with Asymmetric Distance Computation" (Jégou et al., 2011), 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.

The first set of experiments adopts the database and query descriptors from the INRIA Holidays images (Jegou et al., 2008), and assesses the overhead of a staged implementation against a non-staged, equivalent implementation. The second set of experiments uses a test dataset constructed from real data, and provides a more realistic comparison between a staged and a non-staged implementation.

The paper concludes with a brief description of a proof-of-concept implementation of a classified ad retrieval tool based on Jina[2], a framework to build applications leveraging neural search engines, to control the retrieval process, and on the staged HNSW implementation, to index the vector stream.

The rest of this paper is organized as follows. Section 2 covers related work. Section 3 outlines SVS. Section 4 introduces staged IVFADC and the associ-

---

[1]https://redis.io

[2]https://jina.ai

ated experiments. Section 5 describes staged HNSW and the set of experiments with real data. Section 6 briefly describes the proof-of-concept implementation of a classified ad retrieval tool. Finally, Section 7 contains the conclusions.

2 Related Work

2.1 Batch Vector Similarity Search

Similarity search in large scale, high dimensional datasets is an essential feature of several Deep Learning applications (Bengio et al., 2013). 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 et al., 2011). Several algorithms (Muja & Lowe, 2009; Gionis et al., 1999; Datar et al., 2004) 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 main memory still posed a serious limitation for large volumes of data.

The approach proposed in (Jégou et al., 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 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 et al., 2011) is an access method based on product quantization that has been implemented and successfully tested over billions of vectors[3]. An implementation of product quantization that takes advantage of GPUs was also reported in (Johnson et al., 2021).

In more detail, IVFADC uses two quantizers, called a *coarse quantizer* and a *product quantizer*, to index and query vectors, using a set of inverted lists. 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[4] 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.

Ge et al. (2013) introduced an optimized product quantization that minimizes quantization distortions w.r.t. the space decomposition and the quantization codebooks. Cai et al. (2016) described another optimized product quantization scheme which ensures a better subspace partition.

In another direction, 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. 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 k item order 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 (Fu et al., 2019).

Several libraries 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. FAISS[5] is an open-source Python library developed at Meta which offers product quantization and similarity metrics, including IVFADC and IVFlat. FAISS also has a multi-GPU implementation. ScaNN[6] is a similar library developed at Google (Guo et al., 2020). NGT[7] - 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.

---

[3]https://engineering.fb.com/2017/03/29/data-infrastructure/faiss-a-library-for-efficient-similarity-search/

[4]https://github.com/facebookresearch/faiss/wiki/Faiss-indexes

[5]https://github.com/facebookresearch/faiss/wiki/

[6]https://github.com/google-research/google-research/tree/master/scann

[7]https://morioh.com/p/8c38367453ae

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, which is the scenario of this paper.

Milvus[8] is another example of a vector database offering similarity search. It supports, among others, IVFFlat and HNSW.

Table 1 summarizes the main features of some well-known vector indexing libraries and search engines. A detailed comparison of these methods and tools can be found at ANN-Benchmarks[9].

Table 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.

As mentioned in the introduction, Section 4.1 discusses a main memory product quantization implementation of SVS, and assesses the overhead of a staged implementation against a non-staged, equivalent implementation. Section 5 describes the proof-of-concept implementation of staged HNSW that uses Redis with HNSW, and a set of experiments using real data. Therefore, these two sections cover implementations of SVS using product quantization and HNSW, two frequently used vector index methods.

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

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

Yukawa & Amagasa (2021) proposed a method for updating the rotation matrix using SVD-Updating, which can update the singular matrix using low-rank approximations. 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, the proposed family of vector stream similarity search algorithms, 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

3.1 Non-staged Vector Stream Similarity Search

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. CREATEINDEX initializes the index and ADJUSTINDEX hides the details of how the index is adjusted when a new vector is read from the stream.

Algorithm 1 Non-staged ingestion of a stream of vectors V with incremental indexing
---
1: **procedure** NI
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 (which come from a stream). This is one of the problems that the staged methods try to avoid.

3.2 Staged Vector Stream Similarity Search

The family of *staged vector stream similarity search methods*, or briefly *SVS methods*, refers to similarity search methods for vector streams with the following characteristics. An SVS method 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, an SVS method does not depend on having the full set of vectors available beforehand, and it can cope with an unlimited number of vectors.

Members of the SVS family basically differ on the exact vector indexing scheme they use. However, there are 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.

SVS has four major operations:

- INGESTION of a stream of vectors, and indexing and storing the vectors in secondary storage

- RETRIEVAL of vectors by similarity and timestamp, and ranking the retrieved vectors

- DELETION of vectors

- MERGING of indexes

Algorithms 2 and 3 are highly simplified descriptions of the INGESTION operation in pseudocode, for the incremental and deferred alternatives, respectively. They use seven auxiliary procedures: CLOCK, READ, ADDCACHE, CREATEINDEX, ADJUSTINDEX, REINITIALIZEINDEX, and STORE.

CLOCK is a function that returns the current wall clock value.

ADDCACHE adds the newly read vector v to the cache C, along with the current timestamp (which the RETRIEVAL operation uses to help filter the desired vectors). In the incremental alternative, ADJUSTINDEX immediately indexes v, that is, it updates the index I to register v. In the deferred alternative, v is not indexed at this point.

Algorithm 2 Staged ingestion of a stream of vectors V with incremental indexing

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

When the cache becomes full, or a timeout occurs, in the incremental alternative, STORE just stores I, with the time interval T it covers, and REINITIALIZEINDEX reinitializes I. In the deferred alternative, 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. Finally, in both alternatives, STORE moves to secondary storage each vector v in the cache C with the timestamp t when v was read.

Algorithm 4 is again a highly simplified description of the RETRIEVAL operation in pseudocode. The RETRIEVAL 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 RETRIEVAL operation may also search the cache, if its time interval intersects T (not represented in Algorithm 4 for simplicity).

The DELETION operation deletes a specific vector, given its identifier. Once removed, the vector will no longer be retrieved in a search.

Algorithm 3 Staged ingestion of a stream of vectors V with deferred indexing

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

Algorithm 4 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 ← ∅
 3:     for each index I that covers T do
 4:         RETRIEVEVECTORS(q, I; L_c)
 5:         L ← L ∪ L_c
 6:     end for
 7:     Rank L by similarity to q and by timestamp
 8:     Return the ranked list
 9: end procedure
```

Finally, as indexes can get sparse because of deletions, the MERGE operation combines time-adjacent indexes, according to a configurable size heuristic.

3.3 Summary

In summary, the main characteristics of the alternatives for the INGESTION operation are (see Table 2):

- *NI*: the overall cost is dominated by the cost of adjusting the index, since the number of vectors in the stream is not bounded.

- *SI, SD*: at each stage, the cost of adjusting the index is bounded, since the number of vectors is bounded by the cache size.

- *SD*: at each stage, the overhead is not negligible, since an index must be created using the vectors

in the cache; however, the index is specific to the vectors in the cache.

- *SI*: at each stage, the overhead is minimized, since it reduces to reinitializing the index.

Table 2: Abbreviations for the ingestion strategies.

Abbr.	Alg.	Description
NI	1	Non-staged ingestion of a stream of vectors with incremental indexing
SI	2	Staged ingestion of a stream of vectors with incremental indexing
SD	3	Staged ingestion of a stream of vectors with deferred indexing

4 Staged Product Quantization

4.1 Implementations of the INGESTION Operation based on Product Quantization

IVFADC, outlined in Section 2.1, with some adjustments, would provide an implementation of the non-staged INGESTION operation (see Algorithm 1). CREATEINDEX would construct a codebook I upfront from a *learning set* V_0 of vectors (Jégou et al. (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. The disadvantage of this alternative is exactly that it uses an immutable codebook, computed from a learning set of vectors, which may decrease the performance of the retrieval operation over the vectors received from the vector stream.

To circumvent this problem, an online product quantization algorithm (see Section 2.2) that adjusts the codebook could be adopted instead of IVFADC. The disadvantage of this alternative is that adjusting a codebook is an expensive operation, a problem that is exacerbated by the fact that the algorithm has to handle an unbounded vector stream.

IVFADC would also be an alternative to implement the staged INGESTION operation.

Consider first the incremental indexing alternative. One possibility would be to assume that CREATEINDEX constructs a codebook I upfront from a training set and that I is never changed. Then, ADJUSTINDEX 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. Since the codebook I is assumed to be fixed, ADJUSTINDEX would not change I. 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. At the end of each stage, STORE would move the inverted lists to secondary storage, and REINITIALIZEINDEX would simply reinitialize the lists, keeping the original codebook. Hence, the staged method would differ from both the original IVFADC implementation and from PASE.

This implementation would have the disadvantage that it uses a fixed codebook, which might reduce recall, if the codebook is created from a training set of vectors which turns out to be unrelated to the vectors observed in the stream. A second possibility would then be to adopt an online product quantization algorithm to circumvent this problem.

Consider now the deferred alternative. At the end of each stage, CREATEINDEX would construct a different codebook I for the vectors in the cache, rather than a training set. ADJUSTINDEX would then index each vector v in the cache as before. 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.

In summary, the main characteristics of the product quantization alternatives for the INGESTION operation are (see Table 3):

- *IVFADC-NI*: 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, but the inverted lists grow unbounded.

- *IVFADC-SI, IVFADC-SD*: at each stage, the cost of updating the inverted lists is bounded, since the number of entries in the lists is bounded by the cache size.

- *IVFADC-SD*: at each stage, the overhead is not negligible, since a codebook is created using the

vectors in the cache; however, the codebook is specific to the vectors in the cache, which might increase recall.

- *IVFADC-SI*: at each stage, the overhead is minimum when the codebook is fixed, since only a reinitialization of the inverted lists is required; when the codebook is updated by an online product quantization algorithm, the overhead can be non-negligible, however.

Table 3: Abbreviations for the ingestion strategies implemented with product quantization.

IVFADC-NI	IVFADC implementation of NI, the non-staged ingestion of a stream of vectors with incremental indexing
IVFADC-SI	IVFADC implementation of SI, the staged ingestion of a stream of vectors with incremental indexing
IVFADC-SD	IVFADC implementation of SD, the staged ingestion of a stream of vectors with deferred indexing

4.2 Experiments with Staged Product Quantization

The experiments reported in this section assess the performance of the IVFADC implementation of the staged ingestion of a stream of vectors with deferred indexing, referred to as IVFADC-SD in Table 3.

In more detail, the goals of the experiments are:

- *Build cost*: evaluate the cost of IVFADC-SD, for various cache sizes.

- *Query cost*: compare the cost of IVFADC-SD with a baseline when processing query sets.

- *Search quality*: compare the *mean average recall@R* of IVFADC-SD with that of a baseline when processing a set of queries.

The experiments use:

- Base dataset: a random partition of the 1 million INRIA Holidays images into 10 *batches* $B_1,...,B_{10}$

- Query descriptors: from the INRIA Holidays images

The base dataset and the query descriptors are as in (Jégou et al., 2011), except that the base dataset is partitioned into 10 sets.

The partition of the base dataset simulates the use of a cache that holds 100,000 images. That is, the implementation of IVFADC-SD simply processes each batch B_i and trains a codebook with a sample of B_i. Note that this simple strategy greatly facilitates the

Table 4: Recall values of the staged product quantization experiments.

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

experiments since it just simulates IVFADC-SD using a standard implementation of IVFADC.

The baseline is taken as IVFADC applied to the non-partitioned base dataset, trained with a sample of the base dataset (rather than a separate training set as in (Jégou et al., 2011)).

Since the original experiments with this baseline adopted Euclidean distance, it was also adopted here.

The search quality metric is the *mean average recall@R*, as in (Jégou et al., 2011). 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 (Jegou et al., 2008)[10].

In all experiments, the codebooks and the vectors were stored in main memory. The 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 results were:

1. IVFADC-SD and the baseline IVFADC had equivalent total build cost, both in terms of training the codebooks and indexing the data.

2. IVFADC-SD had a significant increase of around 40% in query cost when querying across all 10 batches (the query cost were not detailed in a separated table since they were uniformly 40% higher).

3. Search quality, measured by mean average *recall@R*, saw a slight reduction for lower values of R, but was otherwise similar to the baseline, as observed in Table 4.

These results deserve a few comments. First, note that, if a new batch B_{11} is considered, the baseline IDFADC would have to recompute the codebook. On the other hand, IVFADC-SD would only have the cost associated with processing B_{11}.

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.

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.

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

5 Staged HNSW

This section describes experiments to compare the behavior of staged Hierarchical Navigable Small World (staged HNSW) with a baseline, when the data volume increases. A detailed discussion of implementation alternatives for the INGESTION operation using HNSW would follow along the lines of Section 4.1.

The experiments used data collected from a Brazilian online classified ads company, as follows. 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). Daily, there is an average of 444k approved ads (about 5/sec) entering the platform. The datasets used in these experiments were constructed from approximately 50k, 100k, 250k, 500k, 750k, and 1MM ads collected from 2022/06/01 to 2022/07/10.

For brevity, this section reports experiments that used only the vectors obtained from encoding the ad texts (see Figure 1), all with the same dimension, equal to 768.

Staged HNSW was simulated in Redis[11] with HNSW 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. The experiments divided each dataset into 5 batches.

The baseline used Redis to store the set of vector embeddings resulting from processing the ad texts, without dividing the dataset into batches. Furthermore, the baseline used the "flat" option, that is, Redis processes queries by executing a full sequential scan of the embeddings.

In all experiments, the vectors and the indices were maintained in main memory, which required a much

---

[10]See also http://lear.inrialpes.fr/people/jegou/data.php

[11]https://redis.io

Table 5: 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
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 6: Query processing times (in ms; dataset size = 1,000,000).

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

larger HW configuration. 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.

Table 5 shows the time spent on ingesting the vectors and building the indices in Redis. The lines correspond to the various dataset sizes. The column labeled "**Batch size**" indicates the batch size, which simulates the cache size. The columns labeled "**Batch 0**" through "**Batch 4**" show the time Redis took to ingest and build the HNSW index for the vectors in a given batch. The column labeled "**All batches**" shows the sum of the batch times. The column labeled "**HNSW**" shows the time Redis took to ingest and build the HNSW index for all vectors in a given dataset. Note that the sum of the times to ingest and index the vector for all batches is roughly half of the time to ingest and index the full dataset, which is expected, given the way an HNSW index is built.

Table 6 shows the query processing times. The last column, labeled "**Avg**", discards the outlier values *min* and *max*. Each line corresponds to a search alternative: "**FLAT (k=10)**" indicates that Redis used no index to retrieve the first k=10 vectors closest to Q_i, using Euclidean distance, from the full dataset with 1,000,000 vectors; "**HNSW (k=10)**" indicates that Redis used the HNSW index to retrieve the first k=10 vectors closest to Q_i, using Euclidean distance, from the full dataset with 1,000,000 vectors; and "**HNSW batch i (k=4)**" indicates that Redis used 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.

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.

To assess precision and recall, we selected, from the dataset with 1,000,000 vectors, 10 vectors to play the role of queries. For each query Q_i, we considered as the *relevant vectors* the top-10 vectors retrieved by Redis with the "flat" option from the dataset with 1,000,000 vectors, which amounts to the 10 vectors closest to Q_i, in Euclidian distance, since Redis with the "flat" option performs a full dataset scan.

Euclidean distance was adopted for consistency with the experiments reported in Section 4.2.

Table 7 shows the *mean average precision@k*, for $k = 1, 5, 10$. Lines labeled "**HNSW**" correspond to processing each query Q_i using Redis with HNSW over the full dataset with 1,000,000 vectors. Likewise, lines labeled "**HNSW batches**" correspond to process 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 8 shows the *mean average recall@k*, for $k = 1, 5, 10$, and is similarly organized.

Finally, a closer look at some query examples 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 subscribed to different ads which were copies of each other. Thus, the first ten vectors retrieved were from these copies with a Euclidean distance equal to 0. This suggests deduplicating the ads before constructing the test datasets.

Table 7: Precision values of the query experiments.

		Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Avg
HNSW	**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 batches	**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 8: Recall values of the query experiments.

		Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	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	0.10	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 batches	**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

6 A Classified Ad Retrieval Tool

This section briefly outlines a proof-of-concept classified ad retrieval tool[12], based on Jina, [13], a framework to build applications leveraging neural search engines, to control the retrieval process, and on the staged HNSW implementation introduced in Section 5 to index the vector stream.

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.

Each ad has a key, a name, a brief textual description, and an image. The text and image encoders, as the name implies, transform the text and image of an ad into their vector representations. The vectors resulting from the encodings are stored in Redis, as indicated in Section 5. Each index depends on several parameters, such as the embedding dimension, the metric used to compute the distance between two vectors, and the type of the index.

Following the staged strategy with deferred indexing, the tool buffers 50.000 ads before encoding and storing their text and images.

As an example of the retrieval output, suppose that 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 will return:

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.*********").

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.

7 Conclusions

The main contribution of this paper 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 is continuous in the sense that it does not depend on having the full set of vectors available beforehand, but adapts to the vector stream.

---

[12]Available at https://github.com/BrunoFMSilva/projeto-final-multimodal-clustering

[13]https://jina.ai

This family of algorithms provides an elegant solution to the *vector stream similarity search* problem that does not depend on updating the underlying vector indexing method, which is usually expensive, as pointed out in the related work section. Indeed, the original contribution of the paper 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 paper 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 et al., 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 reasonable search quality. However, SVS can support unbounded vector streams.

The paper 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 testing different configurations by varying the embedding dimensions, the type of the indices, the distance metrics adopted, and some optimization parameters.

As future work, we 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.

REFERENCES

Bengio, Y., Courville, A., & Vincent, P. (2013). Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8), 1798–1828. https://doi.org/10.1109/TPAMI.2013.50

Beyer, K., Goldstein, J., Ramakrishnan, R., & Shaft, U. (1999). When is "nearest neighbor" meaningful? *International conference on database theory*, 217–235. https://doi.org/10.1007/3-540-49257-7_15

Cai, Y., Ji, R., & Li, S. (2016). Dynamic programming based optimized product quantization for approximate nearest neighbor search. *Neurocomputing*, 217, 110–118. https://doi.org/10.1016/j.neucom.2016.01.112

Costa Pereira, J., Coviello, E., Doyle, G., Rasiwasia, N., Lanckriet, G., Levy, R., & Vasconcelos, N. (2014). On the role of correlation and abstraction in cross-modal multimedia retrieval. *Transactions of Pattern Analysis and Machine Intelligence*, 36(3), 521–535. https://doi.org/10.1109/TPAMI.2013.142

Datar, M., Immorlica, N., Indyk, P., & Mirrokni, V. S. (2004). Locality-sensitive hashing scheme based on p-stable distributions. *Proceedings of the twentieth annual symposium on Computational geometry*, 253–262. https://doi.org/10.1145/997817.997857

Fu, C., Xiang, C., Wang, C., & Cai, D. (2019). Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proc. VLDB Endow.*, 12(5), 461–474. https://doi.org/10.14778/3303753.3303754

Ge, T., He, K., Ke, Q., & Sun, J. (2013). Optimized product quantization for approximate nearest neighbor search. *2013 IEEE Conference on Computer Vision and Pattern Recognition*, 2946–2953. https://doi.org/10.1109/CVPR.2013.379

Gionis, A., Indyk, P., Motwani, R., et al. (1999). Similarity search in high dimensions via hashing. *Proc. 25th International Conference on Very Large Data Bases*, 518–529.

Guo, R., Sun, P., Lindgren, E., Geng, Q., Simcha, D., Chern, F., & Kumar, S. (2020). Accelerating large-scale inference with anisotropic vector quantization. *Proc. 37th International Conference on Machine Learning*, ICML'20, 10.

Hameed, I. M., Abdulhussain, S. H., & Mahmmod, B. M. (2021). Content-based image retrieval: A review of recent trends. *Cogent Engineering*, 8(1), 1927469. https://doi.org/10.1080/23311916.2021.1927469

Jegou, H., Douze, M., & Schmid, C. (2008). Hamming embedding and weak geometric consistency for large scale image search. *Computer Vision – ECCV 2008*, 304–317. https://doi.org/10.1007/978-3-540-88682-2_24

Johnson, J., Douze, M., & Jegou, H. (2021). Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(03), 535–547. https://doi.org/10.1109/TBDATA.2019.2921572

Jégou, H., Douze, M., & Schmid, C. (2011). Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1), 117–128. https://doi.org/10.1109/TPAMI.2010.57

Li, X., Yang, J., & Ma, J. (2021). Recent developments of content-based image retrieval (cbir). *Neurocomputing*, 452, 675–689. https://doi.org/10.1016/j.neucom.2020.07.139

Liu, C., Lian, D., Nie, M., & Hu, X. (2020). Online optimized product quantization. *2020 IEEE International Conference on Data Mining (ICDM)*, 362–371. https://doi.org/10.1109/ICDM50108.2020.00045

Malkov, Y. A. & Yashunin, D. A. (2020). Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(4), 824–836. https://doi.org/10.1109/TPAMI.2018.2889473

Muja, M. & Lowe, D. G. (2009). Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP (1)*, 2(331-340), 2. https://doi.org/10.5220/0001787803310340

Xu, D., Tsang, I. W., & Zhang, Y. (2018). Online product quantization. *IEEE Transactions on Knowledge and Data Engineering*, 30(11), 2185–2198. https://doi.org/10.1109/TKDE.2018.2817526

Yang, W., Li, T., Fang, G., & Wei, H. (2020). Pase: Postgresql ultra-high-dimensional approximate nearest

neighbor search extension. *Proc. 2020 ACM SIGMOD International Conference on Management of Data*, 2241–2253. https://doi.org/10.1145/3318464.3386131

Yukawa, K. & Amagasa, T. (2021). Online optimized product quantization for dynamic database using svd-updating. *Database and Expert Systems Applications*, 273–284. https://doi.org/10.1007/978-3-030-86472-9_25

Zeng, D., Yu, Y., & Oyama, K. (2020). Deep triplet neural networks with cluster-cca for audio-visual cross-modal retrieval. *ACM Trans. Multimedia Comput. Commun. Appl.*, 16(3). https://doi.org/10.1145/3387164