Link and code: Fast indexing with graphs and compact regression codes

Matthijs Douze, Alexandre Sablayrolles, Hervé Jégou

▶ **To cite this version:**

Matthijs Douze, Alexandre Sablayrolles, Hervé Jégou. Link and code: Fast indexing with graphs and compact regression codes. CVPR 2018 - IEEE Conference on Computer Vision & Pattern Recognition, Jun 2018, Salt Lake City, United States. pp.3646-3654, 10.1109/CVPR.2018.00384 . hal-01955971

HAL Id: hal-01955971
https://inria.hal.science/hal-01955971

Submitted on 14 Dec 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

arXiv:1804.09996v2 [cs.CV] 27 Apr 2018

Link and code: Fast indexing with graphs and compact regression codes

Matthijs Douze[†], Alexandre Sablayrolles[†,⋆], and Hervé Jégou[†]

[†]Facebook AI Research [⋆]Inria*

Abstract

Similarity search approaches based on graph walks have recently attained outstanding speed-accuracy trade-offs, taking aside the memory requirements. In this paper, we revisit these approaches by considering, additionally, the memory constraint required to index billions of images on a single server. This leads us to propose a method based both on graph traversal and compact representations. We encode the indexed vectors using quantization and exploit the graph structure to refine the similarity estimation.

In essence, our method takes the best of these two worlds: the search strategy is based on nested graphs, thereby providing high precision with a relatively small set of comparisons. At the same time it offers a significant memory compression. As a result, our approach outperforms the state of the art on operating points considering 64–128 bytes per vector, as demonstrated by our results on two billion-scale public benchmarks.

1. Introduction

Similarity search is a key problem in computer vision. It is a core component of large-scale image search [32, 38], pooling [41] and semi-supervised low-shot classification [16]. Another example is classification with a large number of classes [22]. In the last few years, most of the recent papers have focused on compact codes, either binary [10, 18] or based on various quantization methods [26, 11, 3, 44]. Employing a compact representation of vectors is important when using local descriptors such as SIFT [31], since thousands of such vectors are extracted per image. In this case the set of descriptors typically requires more memory than the compressed image itself. Having a compressed indexed representation employing $8-32$ bytes per descriptor was a requirement driven by scalability and practical considerations.

However, the recent advances in visual description have mostly considered description schemes [37, 27, 19] for which each image is represented by an unique vector, typically extracted from the activation layers of a convolutional neural network [5, 42]. The state of the art in image retrieval learns the representation end-to-end [20, 39] such that cosine similarity or Euclidean distance reflects the semantic similarity. The resulting image descriptors consist of no more than a few hundred components.

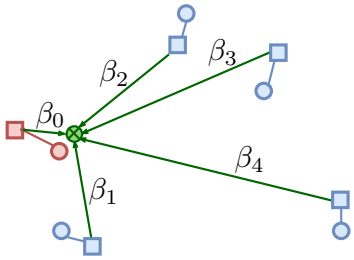

Figure 1. Illustration of our approach: we adopt a graph traversal strategy [34] that maintains a connectivity between all database points. Our own algorithm is based on compressed descriptors to save memory: each database vector (circle) is approximated (square) with quantization. We further improve the estimate by regressing each database vector from its encoded neighbors, which provides an excellent representation basis. The regression coefficients $\boldsymbol{\beta} = [\beta_0, \ldots, \beta_k]$ are selected from a codebook learned to minimize the vector reconstruction error.

In this context, it is worth investigating approaches for nearest neighbor search trading memory for a better accuracy and/or efficiency. An image representation of 128 bytes is acceptable in many situations, as it is comparable if not smaller than the meta-data associated with it and stored in a database. While some authors argue that the performance saturates beyond 16 bytes [8], the best results achieved with 16 bytes on the Deep10M and Deep1B datasets do not exceed 50% recall at rank 1 [7, 15]. While going back to the original vectors may improve the recall, it would require to access a slower storage device, which would be detrimental to the overall efficiency.

One the opposite, some methods like those implemented in FLANN [35] consider much smaller datasets and target high-accuracy and throughput. The state-of-the-art methods implemented in NMSLIB [9] focuses solely on the compro-

*University Grenoble Alpes, Inria, CNRS, Grenoble INP, LJK.

mise between speed and accuracy. They do not include any memory constraint in their evaluation, and compare methods only on small datasets comprising a few millions vectors at most. Noticeably, the successful approach by Malkov *et al.* [33, 34] requires both the original vectors and a full graph structure linking vectors. This memory requirement severely limits the scalability of this class of approaches, which to the best of our knowledge have never been scaled up to a billion vector.

These two points of views, namely compressed-domain search and graph-based exploration, consider extreme sides of the spectrum of operating points with respect to memory requirements. While memory compactness has an obvious practical advantage regarding scalability, we show in Section 3 that HNSW (Hierarchical Navigable Small Worlds [33, 34]) is significantly better than the Inverted Multi-Index (IMI) [6] in terms of the compromise between accuracy and the number of elementary vector comparisons, thanks to the effective graph walk that rapidly converges to the nearest neighbors of a given query.

We aim at conciliating these two trends in similarity search by proposing a solution that scales to a billion vectors, thanks to a limited memory footprint, and that offers a good accuracy/speed trade-off offered by a graph traversal strategy. For this purpose, we represent each indexed vector by i) a compact representation based on the optimized product quantization (OPQ) [17], and ii) we refine it by a novel quantized regression from neighbors. This refinement exploits the graph connectivity and only requires a few bytes by vector. Our method learns a regression codebook by alternate optimization to minimize the reconstruction error.

The contributions of our paper consist of a preliminary analysis evaluating different hypotheses, and of an indexing method employing a graph structure and compact codes. Specifically,

- We show that using a coarse centroid provides a better approximation of a descriptor than its nearest neighbor in a typical setting, suggesting that the first approximation of a vector should be a centroid rather than another point of the dataset [8]. We also show that a vector is better approximated by a linear combination of a small set of its neighbors, with fixed mixing weights obtained by a close-form equation. This estimator is further improved if we can store the weights on a per-vector basis.
- We show that HNSW offers a much better selectivity than a competitive method based on inverted lists. This favors this method for large representations, as opposed to the case of very short codes (8–16 bytes).
- We introduce a graph-based similarity search method with compact codes and quantized regression from neighbors. It achieves state-of-the-art performance on billion-sized benchmarks for the high-accuracy regime.

The paper is organized as follows. After a brief review of related works in Section 2, Section 3 presents an analysis covering different aspects that have guided the design of our method. We introduce our approach in Section 4 and evaluated it in Section 5. Then we conclude.

2. Related work

Consider a set of N elements $\mathcal{X} = \{x_1, \ldots, x_N\} \subset \Omega$ and a distance $d : \Omega \times \Omega \to \mathbb{R}$ (or similarity), we tackle the problem of finding the nearest neighbors $\mathcal{N}_{\mathcal{X}}(y) \subset \mathcal{X}$ of a query $y \in \Omega$, *i.e.*, the elements $\{x\}$ of $\mathcal{X}$ minimizing the distance $d(y, x)$ (or maximizing the similarity, respectively). We routinely consider the case $\Omega = \mathbb{R}^d$ and $d = \ell_2$, which is of high interest in computer vision applications.

Somehow reminiscent of the research field of compression in the 90*s*, for which we have witnessed a rapid shift from lossless to lossy compression, the recent research effort in this area has focused on *approximate* near- or nearest neighbor search [24, 23, 14, 26, 18, 6], in which the guarantee of exactness is traded against high efficiency gains.

Approximate methods typically improve the efficiency by restricting the distance evaluation to a subset of elements, which are selected based on a locality criterion induced by a space partition. For instance Locality Sensitive Hashing (LSH) schemes [12, 1] exploit the hashing properties resulting from the Johnson-Lindenstrauss lemma. Errors occur if a true positive is not part of the selected subset.

Another source of approximation results from compressed representations, which were pioneered by Weber *et al.* [43] to improve search efficiency [43]. Subsequently the seminal work [10] of Charikar on sketches has popularized compact binary codes as a scalability enabler [25, 32]. In these works and subsequent ones employing vector quantization [26], errors are induced by the approximation of the distance, which results in swapped elements in the sorted result lists. Typically, a vector is reduced by principal component analysis (PCA) dimensionality reduction followed by some form of quantization, such as scalar quantization [40], binary quantization [21] and product quantization or its variants [26, 17]. Recent similarity search methods often combine these two approximate and complementary strategies, as initially proposed by Jégou *et al.* [26]. The quantization is hierarchical, *i.e.*, a first-level quantizer produces an approximate version of the vector, and an additional code refines this approximation [28, 3].

The IVFADC method of [26] and IMI [6] are representative search algorithms employing two quantization levels. All the codes having the same first-level quantization code are stored in a contiguous array, referred to as an inverted list, which is scanned sequentially. AnnArbor [8] encodes the vectors w.r.t. a fixed set of nearest vectors. Section 3 shows that this choice is detrimental, and that learning the set of anchor vectors is necessary to reach a good accuracy.

Graph-based approaches. Unlike approaches based on space partitioning, the inspirational NN-descent algorithm [13] builds a knn-graph to solve the all-neighbors problem: the goal is to find the k nearest neighbors in $\mathcal{X}$, w.r.t. d, for each $x \in \mathcal{X}$. The search procedure proceeds by local updates and is not exhaustive, *i.e.*, the algorithm converges without considering all pairs $(x, x') \in \mathcal{X}^2$. The authors of NN-decent have also considered it for the approximate nearest neighbor search.

Yuri Malkov *et al.* [33, 34] introduced the most accomplished version of this algorithm, namely HNSW. This solution selects a series of nested subsets of database vectors, or "layers". The first layer contains only a single point, and the base layer is the whole dataset. The sizes of the layers follow a geometric progression, but they are otherwise sampled randomly. For each of these layers HNSW constructs a neighborhood graph. The search starts from the first layer. A greedy search is performed on the layer until it reaches the nearest neighbor of the query within this layer. That vector is used as an entry point in the next layer as a seed point to perform the search again. At the base layer, which consists of all points, the procedure differs: a bread first search starting at the seed produces the resulting neighbors.

It is important that the graph associated with each subset is *not* the exact knn-graph of this subset: long-range edges must be included. This is akin to simulated annealing or other diversification techniques in optimization: a fraction of the evaluated elements must be far away. In HNSW, this diversification is provided in a natural way, thanks to the long-range links enforced by the upper levels of the structure, which are built on fewer points. However, this is not sufficient, which led Malkov *et al.* to design a "shrinking" operator that reduces a list of neighbors for a vector in a way that does not necessarily keeps the nearest ones.

3. Preliminary analysis

This section presents several studies that have guided the design of the approach introduced in Section 4. All these evaluations are performed on $\mathcal{X} = \text{Deep1M} \subset \mathbb{R}^{96}$, *i.e.*, the first million images of the Deep1B dataset [7].

First, we carry out a comparison between the graph-based traversal of HNSW and the clustering-based hashing scheme employed in IMI. Our goal is to measure how effective a method is at identifying a subset containing neighbors with a minimum number of comparisons. Our second analysis considers different estimators of a vector to best approximate it under certain assumptions, including cases where an oracle provides additional information such as the neighbors. Finally, we carry out a comparative evaluation of different methods for encoding the descriptors in a compact form, assuming that exhaustive search with approximate representations is possible. This leads us to identify appealing choices for our target operating points.

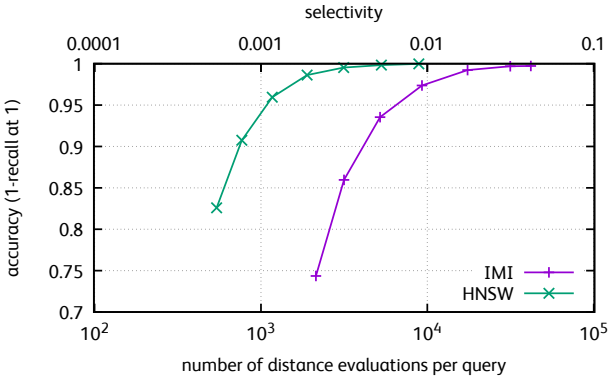

Figure 2. IMI *vs* HNSW: accuracy for a given selectivity.

3.1. Selectivity: HNSW versus IMI

We consider two popular approaches for identifying a subset of elements, namely the multi-scale graph traversal of HNSW [34] and the space partitioning employed in IMI [6], which relies on a product quantizer [26]. Both methods consists of (i) an identification stage, where the query vector is compared with a relatively small set of vectors (centroids or upper level in HNSW); and (ii) a comparison stage, in which most of the actual distance evaluations are performed. For a more direct comparison, we compute the exact distances between vectors. We measure the trade-off between accuracy and the number of distance calculations. This is linearly related to selectivity: this metric [36] measures the fraction of elements that must be looked up.

We select standard settings for this setup: for IMI, we use 2 codebooks of 2^{10} centroids, resulting in about 1M inverted lists. For HNSW, we use 64 neighbors on the base layer and 32 neighbors on the other ones. During the refinement stage, both methods perform code comparisons starting from most promising candidates, and store the k best search results. The number of comparisons after which the search is stopped is a search-time parameter T in both methods. Figure 2 reports the accuracy as a function of the number of distance computations performed for both methods.

The plot shows that HNSW is 5 to 8 times more selective than IMI for a desired level of accuracy. This better selectivity does not directly translate to the same speed-up because HNSW requires many random probes from main memory, as opposed to contiguous inverted lists. Yet this shows that *HNSW will become invariably better than IMI for larger vector representations*, when the penalty of random accesses does not dominate the search time anymore. Figure 3 confirms that HNSW with a scalar quantizer is faster and more accurate than an IMI employing very fine quantizers at both levels. However, this requires 224 bytes per vector, which translates to 50 GB in RAM when including all overheads of the data structure.

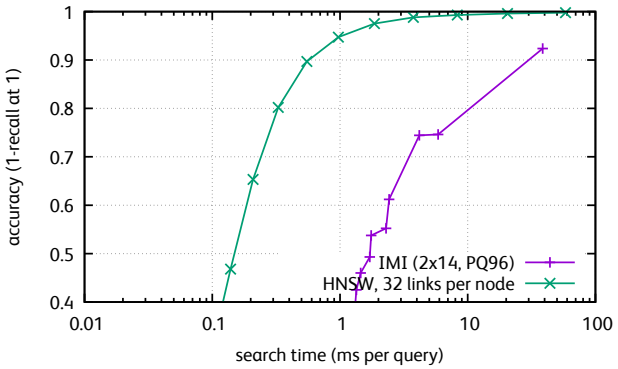

Figure 3. IMI *vs* HNSW on mid-sized dataset (Deep100M): trade-off between speed and accuracy. Both methods use 96-byte encodings of descriptors. The HNSW memory size is larger because of the graph connectivity, see text for details.

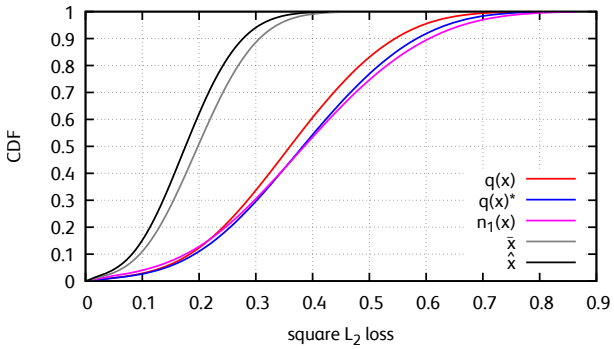

Figure 4. Cumulative probability mass function of the square loss for different estimators of x. $q(x)$: the closest centroid from a codebook learned on $\mathcal{X}$ or $(q(x)^*)$ a distinct set; $n_1(x)$: nearest neighbor in $\mathcal{X}$; $\bar{x}$: weighted average of 8 neighbors; $\hat{x}$: the best estimator from its 8 nearest neighbors.

3.2. Centroids, neighbor or regression?

Hereafter we investigate several ways of getting a coarse approximation[1] of a vector $x \in \mathcal{X}$:

Centroid. We learn by k-means a coarse codebook $\mathcal{C}$ comprising 16k elements. It is learned either directly on $\mathcal{X}$ or using a distinct training set of 1 million vectors. We approximate x by its nearest neighbor $q(x) \in \mathcal{C}$.

Nearest neighbor. We assume that we know the nearest neighbor $n_1(x)$ of x and can use it as an approximation. This choice shows the upper bound of what we can achieve by selecting a single vector in $\mathcal{X}$.

Weighted average. Here we assume that we have access to the $k = 8$ nearest neighbors of x ordered by decreasing distances, stored in matrix form as $\mathbf{N}(x) = [n_1, \ldots, n_k]$. We estimate x as the weighted average

$$\bar{x} = {\beta^\star}^\top \mathbf{N}(x), \tag{1}$$

where $\beta^\star$ is a fixed weight vector constant shared by all elements in $\mathcal{X}$. The close-form computation of $\beta^\star$ is detailled in Section 4.

Regression. Again we use $\mathbf{N}(x)$ to estimate x, but we additionally assume that we perfectly know the optimal regression coefficients $\beta(x)$ minimizing the reconstruction error of x. In other words we compute

$$\hat{x} = \beta(x)^\top \mathbf{N}(x), \tag{2}$$

where $\beta(x)$ is obtained as the least-square minimizer of the over-determined system $\|x - \beta(x)^\top \mathbf{N}(x)\|^2$.

[1]Some of these estimations depend on additional information, for example when we assume that all other vertices of a given database vector x are available. We report them as topline results for the sake of our study.

Figure 4 shows the distribution of the error (square Euclidean distance) for the different estimators. We draw several observations. First, choosing the centroid $q(x)$ in a codebook of 16k vectors is comparatively more effective than taking the nearest neighbor $n_1(x)$ amongst the $64\times$ larger set $\mathcal{X}$ of 1 million vectors. Therefore using vectors of the HNSW upper level graph as reference points to compute a residual vector is not an interesting strategy.

Second, if the connectivity is granted for free or required by design like in HNSW, the performance achieved by $\bar{x}$ suggests that we can improve the estimation of x from its neighbors with no extra storage, if we have a reasonable approximation of $N(x)$.

Third, under the same hypotheses and assuming additionally that we have the parameter $\beta(x)$ for all x, a better estimator can be obtained with Eqn. 2. This observation is the key to the re-ranking strategy introduced in Section 4.

3.3. Coding method: first approximation

We evaluate which vector compression is most accurate *per se* to perform an initial search given a memory budget. Many studies of this kind focus on very compact codes, like 8 or 16 bytes per vector. We are interested in higher-accuracy operating points. Additionally, the results are often reported for systems parametrized by several factors (short-list, number of probes, etc), which makes it difficult to separate the false negatives induced by the coding from those resulting from the search procedure.

To circumvent this comparison issue, we compress and decompress the database vectors, and perform an exhaustive search. All experiments in Table 1 are performed on Deep1M (the 1M first images of the Deep1B dataset). The codebooks are trained on the provided distinct training set. We consider in particular product quantization (PQ [26]) and optimized product quantizer (OPQ [17]). We adopt a

codec	size	accuracy	
	(bytes)	recall@1	recall@10
none	384	1.000	1.000
scalar quantizer	96	0.978	1.000
PQ16x8	16	0.335	0.818
PQ8x16	16	0.394	0.881
PQ2x8+OPQ14x8	16	0.375	0.867
PQ1x16+OPQ14x8	16	0.422	0.899
PQ2x16+OPQ12x8	16	0.421	0.904
PQ2x12+OPQ13x8	16	0.382	0.870
AnnArbor [8]	(*) 16	0.421	
OPQ32x8	32	0.604	0.982
PQ1x16+OPQ30x8	32	0.731	0.997
PQ2x16+OPQ28x8	32	0.713	0.996
PQ2x14+OPQ28x8	32	0.693	0.995
PCA8	32	0.017	0.074

Table 1. HNSW: Exhaustive search in 1M vectors in 96D, with different coding methods. We report the percentage of queries for which the nearest neighbor is among the top1 (resp. top10) results. (*) AnnArbor depends on a parenthood link (4 bytes).

notation of the form PQ16x8 or OPQ14x2, in which the values respectively indicate the number of codebooks and the number of bits per subquantizer.

Additionally, we consider a combination of quantizers exploiting residual vectors [26, 28] to achieve higher-accuracy performance. In this case, we consider a 2-level residual codec, in which the first level is either a vector quantizer with 65536 centroids (denoted PQ1x16 in our notation) or a product quantizer (PQ2x12 or PQ2x14). What remains of the memory budget is used to store a OPQ code for the refinement codec, which encodes the residual vector. Note that IVFADC-based methods and variants like IMI [2] exploit 2-level codecs. Only the data structure differs.

Our results show that 2-level codecs are more accurate than 1-level codecs. They are also more computationally expensive to decode. For operating points of 32 bytes, we observe that just reducing the vectors by PCA or encoding them with a scalar quantizer is sub-optimal in terms of accuracy. Using OPQ gives a much higher accuracy. Thanks to the search based on table lookups, it is also faster than a scalar quantizer in typical settings. For comparison with the AnnArbor method, we also report a few results on 16 bytes per vector. The same conclusions hold: a simple 2-level codec with 65536 centroids (*e.g.*, PQ1x16+OPQ14x8) gets the same codec performance as AnnArbor.

4. Our approach: L&C

This section describes our approach, namely L&C (link and code). It offers a state-of-the-art compromise between approaches considering very short codes (8–32 bytes) and those not considering the memory constraint, like FLANN and HNSW. After presenting an overview of our indexing structure and search procedure, we show how to improve the reconstruction of an indexed vector from its approximate neighbors with no additional memory. Then we introduce our novel refinement procedure with quantized regression coefficients, and details the optimization procedure used to learn the regression codebook. We finally conduct an analysis to discuss the trade-off between connectivity and coding, when fixing the memory footprint per vector.

4.1. Overview of the index and search

Vector approximation. All indexed vectors are first compressed with a coding method independent of the structure. It is a quantizer, which formally maps any vector $x \in \mathbb{R}^d \mapsto q(x) \in \mathcal{C}$, where $\mathcal{C}$ is a finite subset of $\mathbb{R}^d$, meaning that $q(x)$ is stored as a code.

Following our findings of Section 3, we adopt two-level encodings for all experiments. For the first level, we choose a product quantizer of size 2x12 or 2x14 bits (PQ2x12 and PQ2x14), which are cheaper to compute. For the second level, we use OPQ with codes of arbitrary length.

Graph-based structure. We adopt the HSNW indexing structure, except that we modify it so that it works with our coded vectors. More precisely, all vectors are stored in coded format, but the `add` and `query` operations are performed using asymmetric distance computations [26]: the query or vector to insert is not quantized, only the elements already indexed are. We fix the degree of the graphs at the upper levels to $k = 32$, and the size ratio between two graph levels at 30, *i.e.*, there are $30\times$ fewer elements in the graph level 1 than in the graph level 0.

Refinement strategy. We routinely adopt a two-stage search strategy [28]. During the first stage, we solely rely on the first approximation induced by $q(\cdot)$ to select a short-list of potential neighbor candidates. The indexed vectors are reconstructed on-the-fly from their compact codes. The second stage requires more computation per vector is and applied only on this short-list to re-rank the candidates. We propose two variants for this refinement procedure:

- Our **0-byte** refinement does not require any additional storage per vector. It is performed by re-estimating the candidate element from its connected neighbors encoded with $q(x)$. Section 4.2 details this method.
- We refine the vector approximation by using a set of quantized regression coefficients stored for each vector. These coefficients are learned and selected for each indexed vector offline, at building time, see Section 4.3.

4.2. 0-byte refinement

Each indexed vector x is connected in the graph to a set of k other vectors, $g_1(x), \ldots, g_k(x)$, ordered by increas-

ing distance to x. This set can include some of the nearest neighbors of x, but not necessarily. From their codes, we reconstruct x as $q(x)$ and each g_i as $q(g_i(x))$. We define the matrix $\mathbf{G}(x) = [q(x), q(g_1(x)), \ldots, q(g_k(x))]$ stacking the reconstructed vectors. Our objective is to use this matrix to design a better estimator of x than $q(x)$, *i.e.*, to minimize the expected square reconstruction loss. For this purpose, we minimize the empirical loss

$$L(\beta) = \sum_{x \in \mathcal{X}} \|x - \beta^\top \mathbf{G}(x)\|^2. \quad (3)$$

over $\mathcal{X}$. Note that, considering the small set of $k+1$ parameters, using a subset of $\mathcal{X}$ does not make any difference in practice. We introduce the vertically concatenated vector and matrix

$$\mathrm{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix} \quad \text{and} \quad \mathbf{Y} = \begin{bmatrix} \mathbf{G}(x_1) \\ \vdots \\ \mathbf{G}(x_N) \end{bmatrix} \quad (4)$$

and point out that $L(\beta) = \|X - \beta^\top \mathbf{Y}\|^2$. This is a regular least-square problem with a closed-form solution $\beta^\star = \mathbf{Y}^* X$, where $\mathbf{Y}^*$ is the Moore-Penrose pseudo-inverse of $\mathbf{Y}$. We compute the minimizer $\beta^\star$ with a standard regressor. This regression weights are shared by all index elements, and therefore do not involve any per-vector code. A indexed vector is refined from the compact codes associated with x and its connected vectors as

$$\bar{x} = \beta^{\star\top} \mathbf{G}(x). \quad (5)$$

In expectation and by design, $\bar{x}$ is a better approximation of x than $q(x)$, *i.e.*, it reduces the quantization error. It is interesting to look at the weight coefficient in $\beta^\star$ corresponding to the vector $q(x)$ in the final approximation. It can be as small as 0.5 if the quantizer is very coarse: in this situation the quantization error is large and we significantly reduce it by exploiting the neighbors. In contrast, if the quantization error is limited, the weight is typically 0.9.

4.3. Regression codebook

The proposed 0-byte refinement step is granted for free, given that we have a graph connecting each indexed element with nearby points. As discussed in Section 3.2, a vector x would be better approximated from its neighbors if we knew the optimal regression coefficients. This requires to store them on a per-vector basis, which would increase the memory footprint per vector by $4 \times k$ bytes with floating-point values. In order to limit the additional memory overhead, we now describe a method to learn a codebook $\mathcal{B} = \{\beta_1, \ldots, \beta_B\}$ of regression weight vectors. Our objective is to minimize the empirical loss

$$L'(\mathcal{B}) = \sum_{x \in \mathcal{X}} \min_{\beta \in \mathcal{B}} \|x - \beta^\top \mathbf{G}(x)\|^2. \quad (6)$$

Performing a k-means directly on regression weight vectors would optimize the ℓ_2-reconstruction of the regression vector $\beta(x)$, but not of the loss in Eqn. 6. We use k-means only to initialize the regression codebook. Then we use an EM-like algorithm alternating over the two following steps.

1. **Assignment.** Each vector x is assigned to the codebook element minimizing its reconstruction error:

$$\beta(x) = \arg\min_{\beta \in \mathcal{B}} \|x - \beta^\top \mathbf{G}(x)\|^2. \quad (7)$$

2. **Update.** For each cluster, that we conveniently identify by β_i, we find the optimal regression weights

$$\beta_i^\star = \arg\min_{\beta} \sum_{x \in \mathcal{X}: \beta(x) = \beta_i} \|x - \beta^\top \mathbf{G}(x)\|^2 \quad (8)$$

and update $\beta_i \leftarrow \beta_i^\star$ accordingly.

For a given cluster, Eqn. 8 is the same as the one of Eqn. 3, except that the solution is computed only over the subset of vectors assigned to β_i. It is closed-form as discussed earlier.

In practice, as B is relatively small ($B = 256$), we only need a subset of $\mathcal{X}$ to learn a representative codebook $\mathcal{B}$. This refinement stage requires 1 byte per indexed vector to store the selected weight vector from the codebook $\mathcal{B}$.

Product codebook. As shown later in the experimental section, the performance improvement brought by this regression codebook is worth the extra memory per vector. However, the performance rapidly saturates as we increase the codebook size B. This is expected because the estimator $\beta(x)^\top \mathbf{G}(x)$ only spans a $(k+1)$-dimensional subspace of $\mathbb{R}^d$, $k \ll d$. Therefore the projection of x lying in the null space $\ker(\mathbf{G})$ cannot be recovered.

We circumvent this problem by adopting a strategy inspired by product quantization [26]. We evenly split each vector as $x = [x^1; \ldots, x^M]$, where each $x^j \in \mathbb{R}^{d/M}$, and learn a product regression codebook $\mathcal{B}^1 \times \ldots \times \mathcal{B}^M$, *i.e.*, one codebook per subspace. In this case, extending the superscript notation to β and $\mathbf{G}$, the vector is estimated as

$$\hat{x} = [\beta^1(x^1)\mathbf{G}^1(x), \ldots, \beta^M(x^M)\mathbf{G}^M(x)], \quad (9)$$

where $\forall j,\ \beta^j(x^j) \in \mathcal{B}^j$. The set of possible estimators spans a subspace having to up $M \times k$ dimensions. This refinement method requires M bytes per vector.

4.4. L&C: memory/accuracy trade-offs

As discussed earlier, HNSW method is both the fastest and most accurate indexing method at the time being, but its scalability is restricted by its high memory usage. For this reason, it has never been demonstrated to work at a billion-scale. In this subsection, we analyze our algorithm L&C when imposing *a fixed memory budget per vector*. Three factors contribute to the marginal memory footprint:

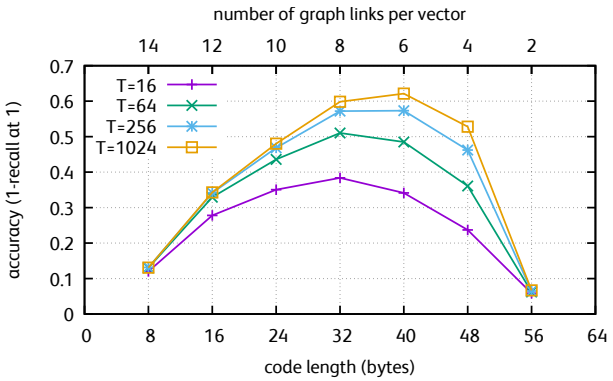

Figure 5. Deep1M: Performance obtained depending on whether we allocate a fixed memory budget of 64 bytes to codes (OPQ codes of varying size) or links. Recall that T is the parameter capping the number of distance evaluations.

- the code used for the initial vector approximation, for instance OPQ32 (32 bytes);
- the number k of graph links per vector (4 bytes per link);
- [*optionally*] the M bytes used by our refinement method from neighbors with a product regression codebook.

L&C Notation. To identify unambiguously the parameter setting, we adopt a notation of the form L6&OPQ40. L6 indicates that we use 6 links per vector in the graph and OPQ40 indicates that we use first encode the vector with OPQ, allocating 40 bytes per vector. If, optionally, we use a regression codebook, we refer to as by the notation M=4 in tables and figures. The case of 0-coding is denoted by M=0.

Coding *vs* Linking. We first consider the compromise between the number of links and the number of bytes allocated to the compression codec. Figure 5 is a simple experiment where we start from the full HNSW representation and reduce either the number of links or the number of dimensions stored for the vectors. We consider all setups reaching the same budget of 64 bytes, and report results for several choices of the parameter T, which controls the total number of comparisons.

We observe that there is a clear trade-off enforced by the memory constraint. The search is ineffective with too few links, as the algorithm can not reache all points. At the opposite side, the accuracy is also impacted by a too strong approximation of the vector, when the memory budget allocated to compression is insufficient. Interestingly, increasing T shifts the optimal trade-off towards allocating more bytes to the code. This means that the neighbors can be reached but require more hops in the graphs.

Coding vectors *vs* regression coefficients. We now fix the number of links to 6 and evaluate the refinement strategy under a fixed total memory constraint. In this case we have a trade-off between the number of bytes allocated to the compression codec and to the refinement procedure.

codec	vector quantization error ($\times 10^3$)		R@1 exhaustive		R@1 T=1024	R@1 T=16384
Deep:	100M	1B	100M	1B	100M	1B
L6&OPQ40	24.3	24.3	0.608	0.601	0.427	0.434
L6&OPQ40 M=0	22.7	22.5	0.611	0.600	0.429	0.435
L6&OPQ36 M=4	21.9	21.5	0.608	0.607	0.428	0.434
L6&OPQ32 M=8	20.0	19.8	0.625	0.612	0.438	0.438

Table 2. Under a constraint of 64 bytes and using $k = 6$ links per indexed vector, we consider different trade-offs for allocating bytes to between codes for reconstruction and neighbors.

The first observation drawn from Table 2 is that the two refinement methods proposed in this section both significantly reduce the total square loss. This behavior is expected for the 0-coding because it is exactly what the method optimizes. However, this better reconstruction performance does not translate to a better recall in this setup. We have investigated the reason of this observation, and discovered that the 0-coding approach gives a clear gain when regressing with the exact neighbors, but those provided by the graph structure have more long-range links.

In contrast, our second refinement strategy is very effective. Coding the regression coefficients with our codebook significantly improves both the reconstruction loss and the recall: the refinement coding based on the graph is more effective than the first-level coding, which is agnostic of the local distribution of the vectors.

5. Experiments

The experiments generally evaluate the search time *vs* accuracy tradeoff, considering also the size of the vector representation. The accuracy is measured as the fraction of cases where the actual nearest neighbor of the query is returned at rank 1 or before some other rank (recall @ rank). The search time is given in milliseconds per query on a 2.5 GHz server with 1 thread. Parallelizing searches with multiple threads is trivial but timings are less reproducible.

5.1. Baselines & implementation

We chose IMI as a baseline method because most recent works on large-scale indexing build upon it [30, 4, 7, 15] and top results for billion-scale search are reported by methods relying on it. We use the competitive implementation of Faiss [29] (in CPU mode) as the IMI baseline. We use the automatic hyperparameter tuning to optimize IMI's operating points. The parameters are the number of visited codes (T), the multiprobe number and the Hamming threshold used to compare polysemous codes [15].

Our implementation of HNSW follows the original NMSLIB version [9]. The most noticeable differences are that (i) vectors are added by batches because the full set of vectors does not fit in RAM, and (ii) the HNSW structure is

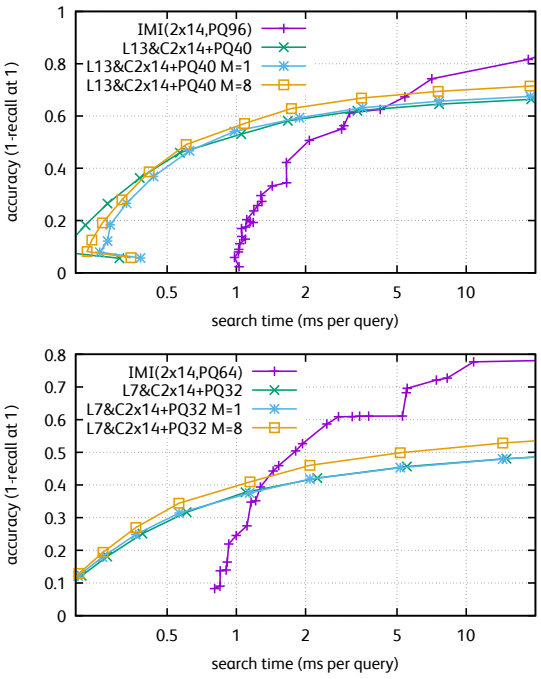

Figure 6. Speed *vs* accuracy on Deep1B (top) and BIGANN (bottom). Timings are measured per query on 1 core.

built layer by layer, which from our observation improve the quality of the graph. Indexing 1 billion vectors takes about 26 hours with L&C: we can add more than 10,000 vectors per second to the index. We refine at most 10 vectors.

For the encodings, we systematically perform a rotation estimated with Optimized Product Quantization (OPQ) to facilitate the encoding in the second level product quantizer.

5.2. Large-scale evaluation

We evaluate on two large datasets widely adopted by the computer vision community. BIGANN [28] is a dataset of 1B SIFT vectors in 256 dimensions, and Deep1B is a dataset of image descriptors extracted by a CNN. Both datasets come with a set of 10,000 query vectors, for which the ground-truth nearest neighbors are provided as well as a set of unrelated training vectors that we use to learn the codebooks for the quantizers. IMI codebooks are trained using 2 million vectors, and the regression codebooks of L&C are trained using 250k vectors and 10 iterations.

Figure 6 compares the operating points in terms of search time *vs* accuracy on Deep1B for encodings that use 96 bytes per vector. For most operating points, our L&S method is much faster, for example 2.5× faster to attain a recall@1 of 50% on Deep1B. The improvement due to the refinement step, *i.e.* the regression from neighborhood, is also significant. It consumes a few more bytes per vector (up to 8).

For computationally expensive operating points, IMI is better for recall@1 because the $4k$=52 bytes spent for links could be used to represent the vertices more accurately.

	R@1	R@10	R@100	time (ms)	bytes
		BIGANN			
Multi-LOPQ [30]	0.430	**0.761**	0.782	8	16
OMulti-D-OADC-L [6]	0.421	0.755	0.782	7	16
FBPQ [4]	0.179	0.523	0.757	1.9	16
	0.186	0.556	**0.894**	9.7	16
Polysemous [15]	0.330		0.856	2.77	16
L7&C32 M=8	**0.461**	0.608	0.613	2.10	72
		Deep1B			
GNO-IMI [7]	0.450	0.8		20	16
Polysemous [15]	0.456			3.66	20
L13&C40 M=8	**0.668**	**0.826**	**0.830**	3.50	108

Table 3. State of the art on two billion-sized datasets.

5.3. Comparison with the state of the art

We compare L&C with other results reported in the literature, see Table 3. Our method uses significantly more memory than others that are primarily focusing on optimizing the compromise between memory and accuracy. However, unlike HNSW, it easily scales to 1 billion vectors on one server. L&C is competitive when the time budget is small and one is interested by higher accuracy. The competing methods are either much slower, or significantly less accurate. On Deep1B, only the polysemous codes attain an efficienty similar to ours, obtained with a shorter memory footprint. However it only attains recall@1=45.6%, against 66.7% for L&C. Considering the recall@1, we outperform the state of the art on BIGANN by a large margin with respect to the accuracy/speed trade-off.

Note, increasing the coding size with other methods would increase accuracy, but would also invariably increase the search time. Considering that, in a general application, our memory footprint remains equivalent or smaller than other meta-data associated with images, our approach offers an appealing and practical solution in most applications.

6. Conclusion

We have introduced a method for precise approximate nearest neighbor search in billion-sized datasets. It targets the high-accuracy regime, which is important for a vast number of applications. Our approach makes the bridge between the successful compressed-domain and graph-based approaches. The graph-based candidate generation offers a higher selectivity than the traditional structures based on inverted lists. The compressed-domain search allows us to scale to billion of vectors on a vanilla server. As a key novelty, we show that the graph structure can be used to improve the distance estimation for a moderate or even null memory budget. As a result, we report state-of-the-art results on two public billion-sized benchmarks in the high-accuracy regime.

Our approach is open-sourced in the Faiss library, see `https://github.com/facebookresearch/faiss/tree/master/benchs/link_and_code`.

References

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for near neighbor problem in high dimensions. In *FOCS*, 2006.

[2] A. Babenko and V. Lempitsky. The inverted multi-index. In *CVPR*, 2012.

[3] A. Babenko and V. Lempitsky. Additive quantization for extreme vector compression. In *CVPR*, 2014.

[4] A. Babenko and V. Lempitsky. Improving bilayer product quantization for billion-scale approximate nearest neighbors in high dimensions. *arXiv preprint arXiv:1404.1831*, 2014.

[5] A. Babenko and V. Lempitsky. Aggregating local deep features for image retrieval. In *CVPR*, 2015.

[6] A. Babenko and V. Lempitsky. The inverted multi-index. *Trans. PAMI*, 2015.

[7] A. Babenko and V. Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *CVPR*, 2016.

[8] A. Babenko and V. Lempitsky. AnnArbor: approximate nearest neighbors using arborescence coding. In *ICCV*, 2017.

[9] L. Boytsov and B. Naidan. Engineering efficient and effective non-metric space library. In *SISAP*, 2013.

[10] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, 2002.

[11] Y. Chen, T. Guan, and C. Wang. Approximate nearest neighbor search by residual vector quantization. *Sensors*, 10(12), 2010.

[12] M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SOCG*, 2004.

[13] W. Dong, M. Charikar, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *WWW*, 2011.

[14] W. Dong, Z. Wang, W. Josephson, M. Charikar, and K. Li. Modeling LSH for performance tuning. In *CIKM*, 2008.

[15] M. Douze, H. Jégou, and F. Perronnin. Polysemous codes. In *ECCV*, 2016.

[16] M. Douze, A. Szlam, B. Hariharan, and H. Jégou. Low-shot learning with large-scale diffusion. *arXiv preprint arXiv:1706.02332*, 2017.

[17] T. Ge, K. He, Q. Ke, and J. Sun. Optimized product quantization for approximate nearest neighbor search. In *CVPR*, 2013.

[18] Y. Gong and S. Lazebnik. Iterative quantization: A procrustean approach to learning binary codes. In *CVPR*, 2011.

[19] Y. Gong, L. Wang, R. Guo, and S. Lazebnik. Multi-scale orderless pooling of deep convolutional activation features. In *ECCV*, 2014.

[20] A. Gordo, J. Almazan, J. Revaud, and D. Larlus. Deep image retrieval: Learning global representations for image search. In *ECCV*, 2016.

[21] A. Gordo and F. Perronnin. Asymmetric distances for binary embeddings. In *CVPR*, 2011.

[22] R. He, Y. Cai, T. Tan, and L. Davis. Learning predictable binary codes for face indexing. *Pattern recognition*, 48, 2015.

[23] P. Indyk. Approximate nearest neighbor algorithms for frechet distance via product metrics. In *SOCG*, 2002.

[24] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC*, 1998.

[25] P. Indyk and N. Thaper. Fast image retrieval via embeddings. In *ICCV Workshop on Statistical and Computational Theories of Vision*, 2003.

[26] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Trans. PAMI*, 33(1), 2011.

[27] H. Jégou, M. Douze, C. Schmid, and P. Pérez. Aggregating local descriptors into a compact image representation. In *CVPR*, 2010.

[28] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: re-rank with source coding. In *ICASSP*, 2011.

[29] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with GPUs. *arXiv preprint arXiv:1702.08734*, 2017.

[30] Y. Kalantidis and Y. Avrithis. Locally optimized product quantization for approximate nearest neighbor search. In *CVPR*, 2014.

[31] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *IJCV*, 2004.

[32] Q. Lv, M. Charikar, and K. Li. Image similarity search with compact data structures. In *CIKM*, 2004.

[33] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45, 2014.

[34] Y. Malkov and D. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *arXiv preprint arXiv:1603.09320*, 2016.

[35] M. Muja and D. G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *Trans. PAMI*, 36, 2014.

[36] L. Paulevé, H. Jégou, and L. Amsaleg. Locality sensitive hashing: a comparison of hash function types and querying mechanisms. *Pattern Recognition Letters*, 31(11), 2010.

[37] F. Perronnin, Y. Liu, J. Sanchez, and H. Poirier. Large-scale image retrieval with compressed Fisher vectors. In *CVPR*, 2010.

[38] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *CVPR*, 2007.

[39] F. Radenović, G. Tolias, and O. Chum. CNN image retrieval learns from bow: Unsupervised fine-tuning with hard examples. In *ECCV*, 2016.

[40] H. Sandhawalia and H. Jégou. Searching with expectations. In *ICASSP*, 2010.

[41] J. Sivic and A. Zisserman. Video Google: A text retrieval approach to object matching in videos. In *ICCV*, 2003.

[42] G. Tolias, R. Sicre, and H. Jégou. Particular object retrieval with integral max-pooling of CNN activations. In *ICLR*, 2016.

[43] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, 1998.

[44] T. Zhang, G.-J. Qi, J. Tang, and J. Wang. Sparse composite quantization. In *CVPR*, 2015.