The Inverted Multi-Index

Artem Babenko [1,2] and Victor Lempitsky [3]
[1] Yandex, [2] National Research University Higher School of Economics
[3] Skolkovo Institute of Science and Technology

Abstract—A new data structure for efficient similarity search in very large datasets of high-dimensional vectors is introduced. This structure called the inverted multi-index generalizes the inverted index idea by replacing the standard quantization within inverted indices with product quantization. For very similar retrieval complexity and pre-processing time, inverted multi-indices achieve a much denser subdivision of the search space compared to inverted indices, while retaining their memory efficiency. Our experiments with large datasets of SIFT and GIST vectors demonstrate that because of the denser subdivision, inverted multi-indices are able to return much shorter candidate lists with higher recall. Augmented with a suitable reranking procedure, multi-indices were able to significantly improve the speed of approximate nearest neighbor search on the dataset of 1 billion SIFT vectors compared to the best previously published systems, while achieving better recall and incurring only few percent of memory overhead.

Image retrieval, Index, Nearest neighbor search, Product Quantization.

1 Introduction

In computer vision, *inverted indices* (inverted files) [1] are widely used for retrieval and similarity search. For a large dataset of visual descriptors, a typical inverted index is built around a *codebook* containing a set of *codewords*, i.e. a representative set of vectors that may be constructed by performing clustering on the initial dataset. An inverted index then stores the list of vectors that lie in the proximity of each codeword (belong to its *Voronoi cell*). The purpose of an inverted index is then to efficiently generate a list of dataset vectors that lie close to any query vector. Given a query, either the closest codeword or a set of few closest codewords are identified. The lists corresponding to those codewords are then concatenated to produce the answer to the query.

Querying the inverted index avoids evaluating distances between the query and every point in the dataset and, thus, provides a substantial speed-up over the exhaustive search. Furthermore, as the index does not need to contain the original dataset vectors to perform the search, the memory footprint of each data point can be reduced significantly, and only useful metadata (e.g. image IDs or heavily compressed original vectors) can be stored in the list entries. Because of these efficiency benefits, inverted indices are widely used within computer vision systems such as image and video search [1] or location identification [2]. More generally, they can be used within any computer vision task that involves fast near(est) neighbor retrieval or kernel density estimation (i.e. image classification [3], [4], understanding [5], image editing [6], etc.).

The efficiency of inverted indices has however certain limitations that begin to show up for very large datasets of vectors (hundreds of million to billions), which computer vision researchers and practitioners are now tackling [7]–[9]. In this scenario, a very fine partition of the search space is desirable to avoid returning excessively large lists in response to the queries or, put differently, to return vectors that are better localized around the query point. Unfortunately, increasing the number of codewords in order to achieve finer partition also increases the query time and the index construction time. While approximate nearest neighbor approaches (e.g. tree codebooks [10] or kd-trees [11]) may be invoked to make this deceleration graceful, these techniques often reduce the accuracy (recall and precision) of the returned candidate lists considerably.

The goal of this paper is to introduce and evaluate a new data structure called the *inverted multi-index* that is in many respects similar to the inverted index and can therefore be used within computer vision systems in a similar way. The advantage of multi-indices is in their ability to produce much finer subdivisions of the search space without increasing the query time and the preprocessing time compared to inverted indices with moderately-sized codebooks (importantly, the relative increase of memory usage for large datasets is also small). Consequently, multi-indices result in faster and more accurate retrieval and approximate nearest neighbor search, especially when dealing with very large scale datasets, while retaining the memory efficiency of standard inverted indices.

In a nutshell, inverted multi-indices are obtained by replacing the vector quantization inside inverted indices with the *product quantization (PQ)* [12]. PQ proceeds by splitting high-dimensional vectors into dimension groups. PQ then effectively approximates each vector as a concatenation of several codewords of smaller dimensionality, coming from several codebooks pretrained for each group of dimensions separately. Following the PQ idea, an inverted multi-index is constructed as a multi-dimensional table. The entries of this table correspond to all possible tuples of codewords from the codebooks corresponding to different dimension groups. This multi-dimensional table replaces a "flat" table containing entries corresponding to codewords of the standard inverted index.

Similarly to a standard inverted index, each entry of a multi-index table corresponds to a part of the original

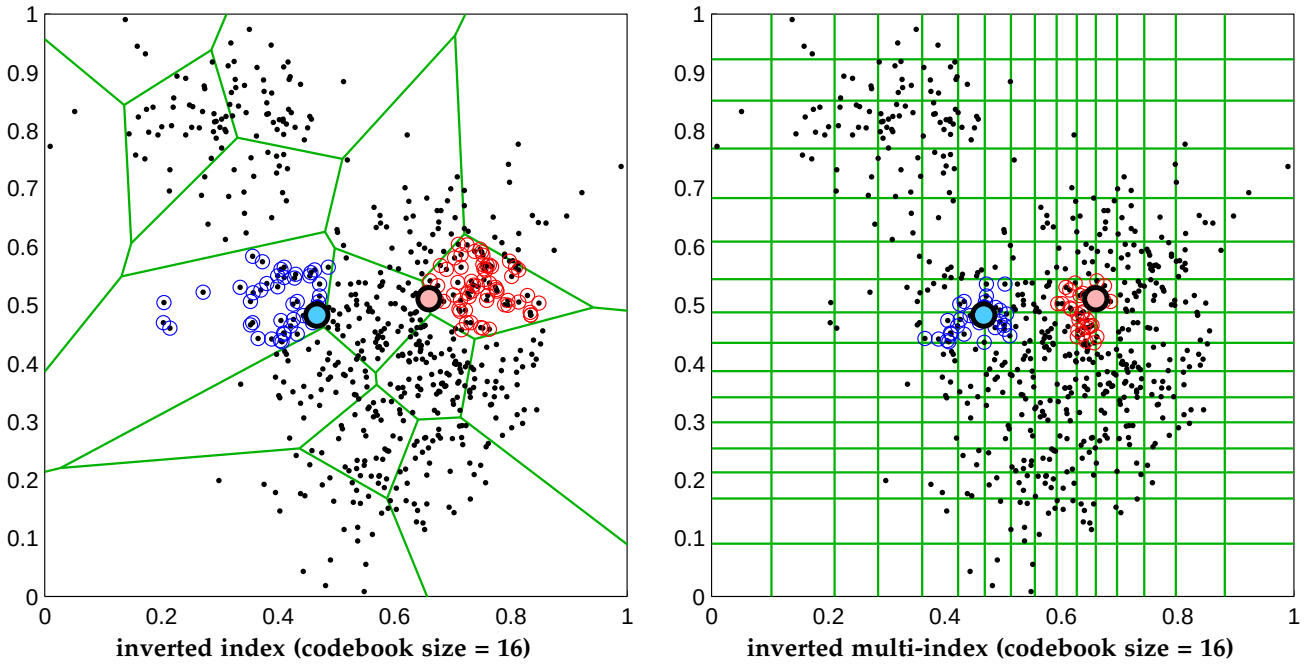


Fig. 1: Indexing the set of 600 points (small black) distributed non-uniformly within the unit 2D square. **Left** – the inverted index based on standard quantization (the codebook has 16 2D codewords; boundaries are in green). **Right** – the inverted multi-index based on product quantization (each of the two codebooks has 16 1D codewords). The number of operations needed to match a query to codebooks is the same for both structures. Two example queries are issued (light-blue and light-red circles). The lists returned by the inverted index (left) contain 45 and 62 words respectively (circled). Note that when a query lies near a space partition boundary (as happens most often in high dimensions) the resulting list is heavily "skewed" and may not contain many of the close neighbors. Note also that the inverted index is not able to return lists of a pre-specified small length (e.g. 30 points). For the same queries, the candidate lists of at least 30 vectors are requested from the inverted multi-index (right) and the lists containing 31 and 32 words are returned (circled). As even such short candidate lists require visiting several nearest cells in the partition (which can be done efficiently via the *multi-sequence algorithm*), the resulting vector sets span the neighborhoods that are much less "skewed" (i.e., the neighborhoods are approximately centered at the queries). In high dimensions, the capability to visit many cells that surround the query from different directions translates into considerably higher accuracy of retrieval and nearest neighbor search.

vector space and contains a list of points that fall within that part. Importantly, we propose a simple and efficient algorithm that produces a sequence of multi-index entries ordered by the increasing distance between the given query vector and the centroid of the corresponding entry. Similarly to standard inverted indices, concatenating the vector lists for a certain number of entries that are closest to the query vector then produces the candidate list.

Crucially, given comparable time budgets for querying the dataset as well as for the initial index construction, inverted multi-indices subdivide the vector space orders of magnitude more densely compared to standard inverted indices (Figure 1). Our experiments demonstrate the advantages resulting from this property, in particular in the context of very large scale approximate nearest neighbor search. We evaluate the inverted multi-index on the BIGANN dataset of 1 billion SIFT vectors recently introduced by Jegou et al. [13] as well as on the "Tiny Images" dataset of 80 million GIST vectors introduced by [9]. We show that as a result of the "extra-fine" granularity, the candidate lists produced by querying multi-indices are more accurate (have shorter lengths and higher probability of containing true nearest neighbors) compared to standard inverted indices. We also demonstrate that in combination with a suitable reranking procedure, multi-indices substantially improve the state-of-the-art approximate nearest neighbor retrieval performance on the BIGANN dataset. Finally, we evaluate the new structure for the task of large scale duplicate image detection.

2 Related Work

The use of inverted indices has a long history in information retrieval [14]. Their use in computer vision was pioneered by Sivic and Zisserman [1]. Since then, a large number of improvements that transfer further ideas from text retrieval (e.g. [15]), improve the quantization process (e.g. [16]), and integrate the query process with geometric verification (e.g. [17]) have been proposed. Many of these improvements can be used in conjunction with inverted multi-indices in the same way as with regular inverted indices.

Approximate near(est) neighbor (ANN) search is a core operation in AI. ANN-systems based on tree-based indices (e.g. [11]) as well as on random projections (e.g. [18]) are often employed. However, the large memory footprint of these methods limits their use to smaller datasets (up to millions of vectors). Recently, lossy compression schemes that admit both compact storage and efficient distance evaluations and are therefore more suitable for large-scale datasets have been developed [19], [20]. Towards this end, binary encoding

schemes (e.g. [21]–[23]) as well as product quantization [12] have brought down both memory consumption and distance evaluation time by order(s) of magnitude compared to manipulating uncompressed vectors, to the point where exhaustive search can be used to query rather large datasets (up to many millions of vectors).

The idea of fast distance computation via product quantization introduced by Jegou et al. [12] has served as a primary inspiration for this work. Our contribution, however, is complementary to that of [12]. In fact, the systems presented by Jegou et al. in [12], [13], [24] use standard inverted indices and, consequently, have to rerank rather long candidate lists when querying very large datasets in order to achieve high recall. Unlike [12], [13], [24], we focus on the use of PQ for indexing and candidate list generation. We also note that while we combine multi-indices with the PQ-based reranking [12] in some of our experiments, one can also employ binary embedding [25] or any other compression/fast distance computation scheme to rerank lists returned by a multi-index (or, depending on the application, omit the reranking altogether).

Since our initial publication [26] two improvements for the nearest neighbor search with multi-index have been suggested. First, Ge et al. [27] have improved the accuracy of the search by replacing the product quantization with the optimized product quantization (OPQ) [28], [29]. Secondly, in Kalantidis et al. [30] and in our report [31], the idea of *local* codebooks that describe the distribution of the points within each cell of a multi-index has been demonstrated to bring further boost to search accuracy.

In general, this work represents a very substantial extension of our previous conference paper [26] with an additional material added from our report [31]. The main algorithmical novelty compared to [26] is a method that enables fast reranking of candidate lists during the approximate nearest neighbor search based on multi-index. The resulting speed improvement is substantial (especially for short codes), while keeping the search accuracy exactly the same. The speed improvement at retrieval time comes at the cost of simple precomputations and lookup tables of limited size stored in memory. On top of that, we evaluate the combination of inverted multi-indices and principal component analysis (PCA), include a detailed comparison between the *second-order* and the *fourth-order* inverted multi-indices, and evaluate inverted multi-indeces on the task of near-duplicate detection. Finally, for the sake of completeness, in the experimental section we evaluate the improvements of the inverted multi-index-based nearest neighbor search discussed above.

3 The Inverted Multi-Index

The structure of the inverted multi-index. We now explain how an inverted multi-index is organized. Along the way, we will compare the analogous parts between inverted multi-indices and standard inverted indices.

We assume that a large collection $\mathcal{D}$ of N M-dimensional vectors $\mathcal{D} = \{\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_N\}$, $\mathbf{p}_i \in \mathcal{R}^M$ is given. The construction of a standard inverted index then starts with learning a codebook $\mathcal{W}$ of K M-dimensional vectors $\mathcal{W} = \{\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_K\}$ via a k-means algorithm. The initial dataset is then split into K *lists* $\mathcal{W}_1, \mathcal{W}_2, \dots, \mathcal{W}_K$, where each list $\mathcal{W}_i$ contains all vectors that fall in its Voronoi cell in $\mathcal{R}^M$, i.e. $\mathcal{W}_i = \{\mathbf{p} \in \mathcal{D} | i = \arg\min_j d(\mathbf{p}, \mathbf{w}_j)\}$. Here, d is a distance measure in $\mathcal{R}^M$. In practice, each list $\mathcal{W}_i$ can be represented in memory as a contiguous array, where each entry may contain the compressed version of the initial vector (which is useful for reranking) and typically some metadata associated with the vector (e.g. the class label or the ID of the image that the visual descriptor $\mathbf{p}$ was sampled from).

Following the product quantization idea [12], the inverted multi-index is organized around splitting the M input dimensions into several dimension blocks. The number of blocks affects the accuracy of retrieval and its speed. In previous works where PQ was used for compression and fast distance evaluation, the best trade-off was achieved for 8 or so blocks [12], [13], [24]. In the multi-index case, however, it is optimal to split dimensions in just two blocks, at least for the characteristic scales considered in our evaluation and assuming that the accuracy and low query time are more important than low index construction time. We comment more on the choice of the number of blocks below. For the time being, to simplify the explanation we discuss how a multi-index can be built for the case of splitting vectors into two halves. Where required, we refer to this case as the *second-order* inverted multi-index. It will be evident how to generalize the proposed algorithms to *higher-order* inverted multi-indices (which split vectors into more than two dimension groups).

Let $\mathbf{p}_i = [\mathbf{p}_i^1 \ \ \mathbf{p}_i^2]$ be the decomposition of a vector $\mathbf{p}_i \in \mathcal{R}^M$ from the dataset into two halves, where $\mathbf{p}_i^1 \in \mathcal{R}^{\frac{M}{2}}, \mathbf{p}_i^2 \in \mathcal{R}^{\frac{M}{2}}$. As in the case of other PQ-based systems, inverted multi-indices perform better when the correlation between $\mathcal{D}^1 = \{\mathbf{p}_i^1\}$ and $\mathcal{D}^2 = \{\mathbf{p}_i^2\}$ is lower and the amount of variance within $\mathcal{D}^1$ and $\mathcal{D}^2$ are closer to each other. For SIFT-vectors, splitting them directly into halves seems to be a near-optimal strategy, while in other cases one can regroup the dimensions to reduce the correlation or multiply all vectors by a random orthogonal matrix to balance the variances between the halves [12], [24].

The PQ codebooks for the inverted multi-index are obtained via independent k-means clustering of the sets $\mathcal{D}^1$ and $\mathcal{D}^2$ independently, producing the codebooks $\mathcal{U} = \{\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_K\}$ for the first half and $\mathcal{V} = \{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_K\}$ for the second half of dimensions[1]. We then perform the product quantization of the dataset vectors, so that the K^2 lists corresponding to all possible pairs of codewords $(\mathbf{u}_i, \mathbf{v}_j), i = 1 \dots K, j = 1 \dots K$ are created. We denote each of the K^2 lists as $\mathcal{W}_{ij}$. Each point $\mathbf{p} = [\mathbf{p}^1 \ \mathbf{p}^2]$ is assigned to the closest point $[\mathbf{u}_i \ \mathbf{v}_j]$, so that:

$$\mathcal{W}_{ij} = \{\mathbf{p} = [\mathbf{p}^1 \ \mathbf{p}^2] \in \mathcal{D} \mid i = \arg\min_k d_1(\mathbf{p}^1, \mathbf{u}_k) \wedge j = \arg\min_k d_2(\mathbf{p}^2, \mathbf{v}_k)\} . \quad (1)$$

Note that the "catchment area" of each list $\mathcal{W}_{ij}$ is now a Cartesian product of the two Voronoi cells in $\mathcal{R}^{\frac{M}{2}}$ spaces. In (1), the distance measures d_1 and d_2 in $\mathcal{R}^{\frac{M}{2}}$ are induced by d, so that $\forall \mathbf{a}, \mathbf{b} : \ d(\mathbf{a}, \mathbf{b}) = d_1(\mathbf{a}^1, \mathbf{b}^1) + d_2(\mathbf{a}^2, \mathbf{b}^2)$. The

1. We have deliberately chosen different letters $\mathbf{u}$ and $\mathbf{v}$ in the notation of the two sub-codebooks, to emphasize that they are learned separately and that $\mathbf{w}_i \neq [\mathbf{u}_i \ \mathbf{v}_i]$.

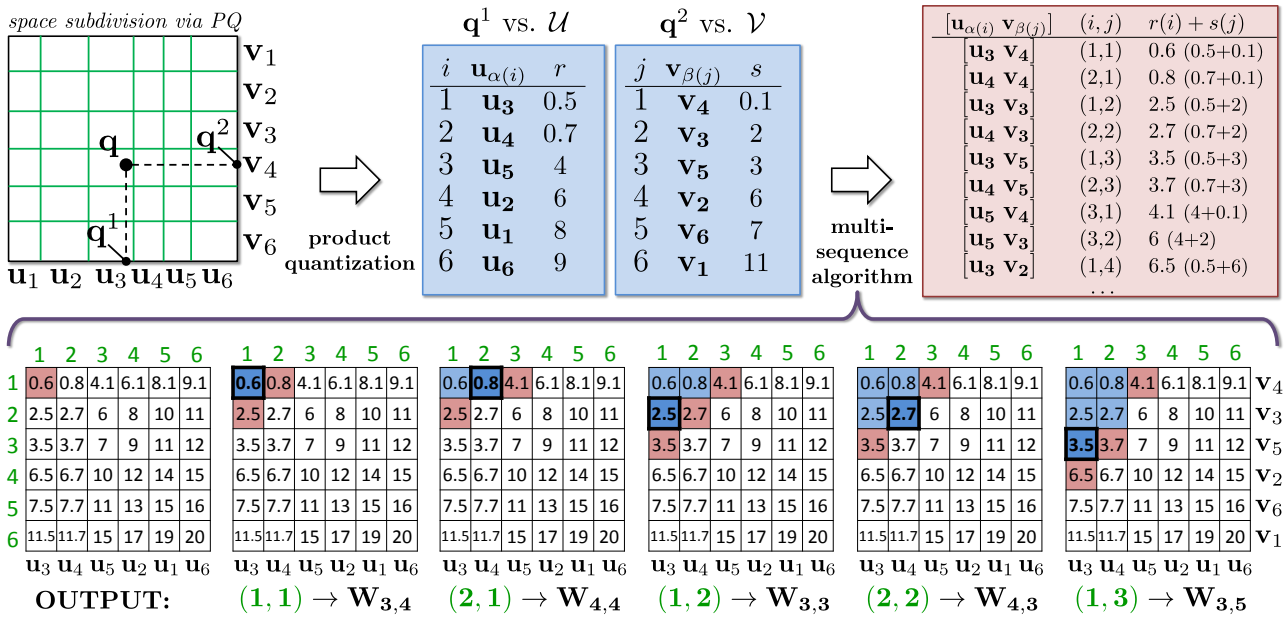


Fig. 2: Top – The overview of the query process within the inverted multi-index. First, the two halves of the query $\mathbf{q}^1$ and $\mathbf{q}^2$ are matched w.r.t. sub-codebooks $\mathcal{U}$ and $\mathcal{V}$ to produce the two sequences of codewords ordered by the distance (denoted r and s) from the respective query half. Then, those sequences are traversed with the multi-sequence algorithm that outputs the pairs of codewords ordered by the distance from the query. The lists associated with those pairs are concatenated to produce the answer to the query. **Bottom** – The first iterations of the multi-sequence algorithm in this example. Red denotes pairs in the priority queue, blue indicates traversed pairs (the pair traversed at the current iteration is emphasized). Green numbers correspond to pair indices (i and j), while black symbols give original codewords ($\mathbf{u}_{\alpha(i)}$ and $\mathbf{v}_{\beta(j)}$). The numbers in entries are the distances $r(i)+s(j) = d\left(\mathbf{q}, [\mathbf{u}_{\alpha(i)}\ \mathbf{v}_{\beta(j)}]\right)$.

simplest and most important case is setting d, d_1, and d_2 to be squared Euclidean norms in respective spaces, so that the resulting multi-index can be used to retrieve points with low Euclidean distance from the query. We briefly discuss alternative distances below.

Querying the multi-index. Given a query $\mathbf{q} = [\mathbf{q}^1\ \mathbf{q}^2] \in \mathcal{R}^M$ and a desired candidate list length $T \ll N$, an inverted multi-index allows to generate a list of T (or slightly more) points from $\mathcal{D}$ that tend to be close to $\mathbf{q}$ with respect to the distance d. This is achieved via identifying a sufficient number of codeword pairs $[\mathbf{u}_i\ \mathbf{v}_j]$ that are closest to $\mathbf{q}$ in $\mathcal{R}^M$ and concatenating their lists $\mathcal{W}_{ij}$. Finding those $[\mathbf{u}_i\ \mathbf{v}_j]$ is performed in two stages (Figure 2-top).

On the first stage, $\mathbf{q}^1$ and $\mathbf{q}^2$ are independently matched to corresponding codebooks. Thus, for $\mathbf{q}^1$ and $\mathbf{q}^2$ the L nearest neighbors among $\mathcal{U}$ and $\mathcal{V}$ respectively are identified (where $L < K$ depends on the specified T). As the size of $\mathcal{U}$ and $\mathcal{V}$ is typically not large (thousands of vectors), exhaustive search can be used. Denote with $\alpha(k)$ the index of the kth closest neighbor to $\mathbf{q}^1$ in $\mathcal{U}$ (i.e. $\mathbf{u}_{\alpha(1)}$ is the nearest neighbor to $\mathbf{q}^1$ in $\mathcal{U}$, $\mathbf{u}_{\alpha(2)}$ is the second closest, etc.). Similarly, denote with $\beta(k)$, the index of the kth closest neighbor to $\mathbf{q}^2$ in $\mathcal{V}$. Also, denote with $r(k)$ and $s(k)$ the distances from $\mathbf{q}^1$ and $\mathbf{q}^2$ to $\mathbf{u}_{\alpha(k)}$ and $\mathbf{v}_{\beta(K)}$ respectively, i.e. $r(k) = d_1\left(\mathbf{q}^1, \mathbf{u}_{\alpha(k)}\right)$ and $s(k) = d_2\left(\mathbf{q}^2, \mathbf{v}_{\beta(k)}\right)$.

On the second stage, given the two monotonically increasing sequences $r(1), r(2), \ldots, r(L)$ and $s(1), s(2), \ldots, s(L)$, we traverse the set of pairs $\{(r(i), s(j)) \,|\, i = 1 \ldots L,\ j = 1 \ldots L\}$ in the order of the increasing sum $r(i)+s(j)$ (which equals $d(\mathbf{q}, [\mathbf{u}_{\alpha(i)}\ \mathbf{v}_{\beta(j)}])$). In this way, the centroids $[\mathbf{u}_{\alpha(i)}\ \mathbf{v}_{\beta(j)}]$ are visited in the order of increasing distance from $\mathbf{q}$. The traversal starts from the pair $(1, 1)$ naturally corresponding to the cell around the centroid $[\mathbf{u}_{\alpha(1)}\ \mathbf{v}_{\beta(1)}]$, which the query falls into. During the traversal, the lists $\mathcal{W}_{\alpha(i)\,\beta(j)}$ are concatenated, until the length of the answer exceeds the predefined length T, at which point the traversal stops.

We propose an algorithm to perform such a traversal (Figure 2-bottom, Figure 3). This *multi-sequence* algorithm is based around a priority queue of index pairs (i, j), where the priority of each pair is defined as $-(r(i) + s(j)) = -d\left(\mathbf{q}, [\mathbf{u}_{\alpha(i)}\ \mathbf{v}_{\beta(j)}]\right)$. The queue is initialized with a single pair $(1, 1)$. At each subsequent step t, the pair (i_t, j_t) with top priority (lowest distance from $\mathbf{q}$) is popped from the queue and considered traversed (the associated list $\mathcal{W}_{\alpha(i)\,\beta(j)}$ is added to the output list). The pairs $(i_t + 1, j_t)$ and $(i_t, j_t + 1)$ are then considered for the insertion into the priority queue. The pair $(i_t + 1, j_t)$ is inserted into the queue if its other preceding pair $(i_t + 1, j_t - 1)$ has also been traversed (or if j_t=1). Similarly, the pair $(i_t, j_t + 1)$ is inserted into the queue if its other preceding pair $(i_t - 1, j_t + 1)$ has also been traversed (or if i_t=1). The idea is that each pair is inserted only once when both of its preceding pairs are traversed.

The multi-sequence algorithm produces a sequence of pairs (i, j), whose lists $\mathcal{W}_{i,j}$ are accumulated into the query response. One can prove the correctness of the algorithm:

Corollary 1 (correctness): the multi-sequence algorithm produces the sequence of pairs in the order of increasing $r(i) + s(i)$ and will eventually visit every pair in $\{1 \ldots L\} \otimes \{1 \ldots L\}$.

Proof: We prove the corollary 1 in two steps. First, we

prove that the algorithm will visit all cells in the L-by-L table (*completeness* – given that long enough candidate list is requested) and then we prove that it will visit the cells in the right order (*monotonicity*).

Completeness: We prove the completeness by induction on the value of sum $(i+j)$ for a fixed value of L. The base of induction, i.e. the case $i+j=2$ is trivial as the cell $(1,1)$ is always traversed at the first step. Assume that all cells $(i,j), i+j \leq k$ will be traversed. Consider a cell (i,j) with $i+j=k+1$. Both its predecessors $(i-1,j)$ and $(i,j-1)$ will be traversed by the assumption. Then, after the traversal of the second predecessor the cell (i,j) will be pushed into the queue and eventually popped from the queue (given that long enough candidate list is requested). Thus the induction step is proved and the completeness is verified.

Monotonicity: Let us now show that for any two cells (i_1,j_1), (i_2,j_2) such that (i_2,j_2) was traversed immediately after (i_1,j_1), the monotinicity holds, i.e. $r(i_1)+s(j_1) \leq r(i_2)+s(j_2)$.

Assume the contrary, i.e. $r(i_1)+s(j_1) > r(i_2)+s(j_2)$. This would mean that (i_2,j_2) was pushed into the priority queue after (i_1,j_1) was traversed (otherwise the algorithm would have popped (i_2,j_2) from the priority queue first). However, after the traverse of (i_1,j_1) algorithm can push into the queue only either (i_1+1,j_1) or (i_1,j_1+1) (or both). As $r(i+1) \geq r(i)$ and $s(i+1) \geq s(i)$, the initial assumption $r(i_1)+s(j_1) > r(i_2)+s(j_2)$ cannot hold for any of those two cases. ■

Regarding the efficiency of the algorithm, one can prove that the queue within the algorithm grows slow enough:

Corollary 2: at the tth step of the algorithm, when t pairs have been output, the priority queue is no longer than $0.5+\sqrt{2t+0.25}$.

Proof: Let us estimate the minimum number of cells that the multi-sequence algorithm has to traverse to get a priority queue of size q. Consider the number of cells traversed in each row (denote them n_i). It is easy to see that a) n_i is monotonically non-increasing; b) each row has at most one cell in the priority queue (for the same reason), c) if $n_i = n_{i+1}$ than the row $i+1$ has no cells in the priority queue. All three statements follows from the fact that each cell can be added to the queue (or traversed) only after all of its predecessors, i.e. all cells with both coordinates smaller or equal to a given one, have been traversed.

Therefore, to get q cells in the priority queue, there should be at least $q-1$ non-empty rows with total number of traversed cells equals $1+2+\cdots+(q-1) = \frac{q(q-1)}{2}$. Therefore, $\frac{q(q-1)}{2} \leq t$, where t is the number of steps (=number of traversed cells). Solving this quadratic inequality gives the bound. ■

Inverted index vs. inverted multi-index. Let us now discuss the relative efficiency of the two indexing structures, given the same codebook size K. In this situation, the induced subdivision of the space is very different for the standard inverted index and for the inverted multi-index (Figure 1). In particular, the standard index maintains K lists that correspond to the space subdivision into K cells, while the multi-index maintains K^2 lists corresponding to a much finer subdivision of the space. While the lengths of the cell lists within the inverted index tend to be somewhat balanced (due to the nature of the k-means algorithm), the distribution of list lengths within the multi-index is highly non-uniform. In particular, there are lots of empty lists that correspond to $\mathbf{u}_i$ and $\mathbf{v}_j$ that never co-occur together (e.g. cells in the bottom-right corner in Figure 1-right). Still, as will be revealed in the experiments, despite a highly non-uniform distribution of list lengths, inverted multi-indices enjoy a large boost in retrieval accuracy due to higher sampling density.

Furthermore, despite the increase in the subdivision density, matching a query with codebooks for both structures requires the same number of operations. Thus, in the inverted multi-index case one has to compute the K distances between M-dimensional vectors, while in the multi-index case $2K$ distances between $M/2$-dimensional vectors are computed (while the number of the scalar operations is the same, vector instructions on modern CPUs can make the matching moderately faster in the inverted index case). Querying the multi-index also incurs an overhead in computational cost due to the use of the multi-sequence algorithm. In our experiments, we however observed (in Section 5) that the overhead was small compared to the quantization cost even for rather long list lengths T.

The use of the inverted multi-index also incurs a memory overhead, as it has to maintain K^2 rather than K lists. However, the joint length of all lists remains the same (as the number of entries equals the total number of vectors i.e. N). Therefore, given that all lists are stored contiguously as

Algorithm 3.1: MULTI-SEQUENCE ALGORITHM()

INPUT : $r(:),\ s(:)$ *% The two input sequences*
OUTPUT : out(:) *% The sequence of index pairs*
% Initialization:
out $\leftarrow \emptyset$
traversed (1:length(r), 1:length(s)) $\leftarrow$ **false**
pqueue $\leftarrow$ new PriorityQueue
pqueue.push ($(1,1)$, $r(1)+s(1)$)
% Traversal:
repeat
  $((i,j),d) \leftarrow$ pqueue.pop()
  traversed$(i,j) \leftarrow$ **true**
  out $\leftarrow$ out $\cup \{(i,j)\}$
  if $i <$ length(r) **and** (j=1 **or** traversed$(i+1,j-1)$)
    then pqueue.push ($(i+1,j)$, $r(i+1)+s(j)$)
  if $j <$ length(s) **and** (i=1 **or** traversed$(i-1,j+1)$)
    then pqueue.push ($(i,j+1)$, $r(i)+s(j+1)$)
until (enough traversed)

Fig. 3: The pseudocode for the multi-sequence algorithm. In our implementation, the iterations are stopped whenever the total number of elements within the entries corresponding to the output pairs of indices exceeds a user-prespecified length. Here, we give the variant of the multi-sequence algorithm for combining two sequences. The generalization to the "higher-order" case (e.g. merging four sequences) is straightforward.

a large array, maintaining each list $\mathcal{W}_{ij}$ effectively requires one integer (that contains the starting location of the list within the large array). Within our experimental setting of $N = 10^9$ and $K = 2^{14}$, this overhead amounts to one byte per dataset vector (4 bytes*K/N). Such overhead is small compared to several bytes of meta-data and/or compressed vector that are typically stored for each instance.

Coming back to higher-order multi-indices, which split vectors into more than two dimension groups, our experiments suggest that while they result in much smaller quantization times (for the same subdivision densities), their memory overheads grow quite rapidly with K and so does non-uniformity of list lengths and the numbers of empty cells in the index. This memory inefficiency limits the usage of such "higher-order" multi-indices to small values of K, where the accuracy of retrieval is limited. Overall, in our experiments, second-order multi-indices proved to be a sweet spot between inverted indices (low memory overhead, large quantization times for sufficient subdivision density) and higher-order multi-indices (high memory overhead, low quantization time). The use of the latter, however, might be justified when small pre-processing times are required, as the time required to product quantize all dataset vectors during higher-order multi-index construction is much smaller (due to lower K).

4 Approximate Nearest Neighbor Search with Inverted Multi-Index

The most important application of the inverted multi-index is the large-scale approximate nearest neighbor search (ANN). Indeed, as an indexing structure (e.g. a multi-index) return short lists of vectors that are close to a query vector, one can *rerank* the vectors based on some additional information stored about each vector. Following [12], we consider two possibilities to encode the information about each vector.

First, we can store a PQ-compressed version of each vector (we call this variant *Multi-ADC*). In the case of Multi-ADC, the multi-index is used to return the candidate lists of PQ-compressed points. A standard asymmetric distance computation (ADC) is then used to rerank the returned list by computing the distances between the query and the returned compressed representations.

Alternatively, we store a PQ-compressed version of the residual displacement between the vector x and the closest centroid $[c_i^1\ c_j^2]$. This is analogous to the IVFADC system of [12], except that the inverted index is replaced with the inverted multi-index. Thus, the second-order Multi-D-ADC system is built around the *coarse* codebooks C^1, C^2 that define the multi-index structure. In addition, Multi-D-ADC includes the *fine* codebooks $R_1, \dots, R_M$, that are used to PQ-compress the displacements between the dataset points and the cell centroids. We assume that the codebooks $R_1, \dots, R_{\frac{M}{2}}$ are used to encode the first half the displacements dimensions, and the codebooks $R_{\frac{M}{2}+1}, \dots, R_M$ encode the second half.

In general, for the same number of extra bytes, Multi-D-ADC leads to higher recall than Multi-ADC, because residual displacements have smaller magnitudes than the original points and hence allow less lossy PQ compression. On the other hand, Multi-ADC is faster than our initial implementation of Multi-D-ADC [26], since it allows efficient pre-computation of a single look-up table for the ADC computation. Below, we describe a method that speeds up the Multi-D-ADC system considerably without affecting the returned results.

Generally, in the case of the IFVADC system, the reranking of PQ-compressed candidates is highly efficient. In each visited cell the displacement of a query from the cell centroid is calculated and the lookup tables for fast ADC [12] procedure are then computed, which permit fast distance computations between the query displacement vector and the displacement vectors of points stored in that cell. In the case of Multi-D-ADC, this approach is inefficient since most of the multi-index cells contain few points and the cost of precomputing the ADC tables is not justified. Here (and in the report [31]), we describe a trick that allows to overcome this difficulty.

Consider now the case of the Multi-D-ADC and the Euclidean distance between query $q \in \mathbf{R}^D$ and a dataset point x belonging to a cell W_{ij} with the centroid $[c_i^1, c_j^2]$. Let the displacement of x from the cell centroid to have the PQ-code $[r_1, \dots, r_M]$. With such a code x is effectively approximated by a sum:

$$x \approx \begin{pmatrix} c_i^1 \\ c_j^2 \end{pmatrix} + \begin{pmatrix} r_1 \\ \vdots \\ r_M \end{pmatrix} \tag{2}$$

Then the distance from the query to the point is approximated using (2):

$$\|q - x\|^2 \approx \left\| q - \begin{pmatrix} c_i^1 \\ c_j^2 \end{pmatrix} - \begin{pmatrix} r_1 \\ \vdots \\ r_M \end{pmatrix} \right\|^2 = \tag{3}$$

$$||q||^2 - 2\left\langle q, \begin{pmatrix} c_i^1 \\ c_j^2 \end{pmatrix} \right\rangle - 2\left\langle q, \begin{pmatrix} r_1 \\ \vdots \\ r_M \end{pmatrix} \right\rangle + \left\| \begin{pmatrix} c_i^1 \\ c_j^2 \end{pmatrix} + \begin{pmatrix} r_1 \\ \vdots \\ r_M \end{pmatrix} \right\|^2$$

As usual, we can precompute the dot-products of the query subvectors with the codewords both from the coarse codebooks C^1, C^2 and the fine codebooks $R_1, \dots, R_M$. These displacements are stored in lookup tables and reused in each cell during the calculation of the terms $\left\langle q, \begin{pmatrix} c_i^1 \\ c_j^2 \end{pmatrix} \right\rangle$ and $\left\langle q, \begin{pmatrix} r_1 \\ \vdots \\ r_M \end{pmatrix} \right\rangle$. Given that dot-products are precomputed, for each dataset point the calculation of these terms can be done in $O(M)$ operations.

In addition, we note that the terms $\left\| \begin{pmatrix} c_i^1 \\ c_j^2 \end{pmatrix} + \begin{pmatrix} r_1 \\ \vdots \\ r_M \end{pmatrix} \right\|^2$ are query-independent. They can then be precomputed in

advance and also stored within the lookup tables. Due to the nice properties of PQ this term can be further simplified:

$$\left\| \begin{pmatrix} c_i^1 \\ c_j^2 \end{pmatrix} + \begin{pmatrix} r_1 \\ \vdots \\ r_M \end{pmatrix} \right\|^2 = \left\| c_i^1 + \begin{pmatrix} r_1 \\ \vdots \\ r_{\frac{M}{2}} \end{pmatrix} \right\|^2 + \left\| c_j^2 + \begin{pmatrix} r_{\frac{M}{2}+1} \\ \vdots \\ r_M \end{pmatrix} \right\|^2 = \tag{4}$$

$$\left\|c_i^1\right\|^2 + \left\|c_j^2\right\|^2 + \sum_{k=1}^{M} \|r_k\|^2 + 2\sum_{k=1}^{\frac{M}{2}} \langle c_i^1, r_k \rangle + 2 \sum_{k=\frac{M}{2}+1}^{M} \langle c_j^2, r_k \rangle$$

Thus, it is enough to precompute and store the squared norms of the codewords and the dot-products of the coarse-level and the fine-level codewords. Given these values the calculation of the query-independent square term can be also performed in $O(M)$ operations. As a result, all terms within the distance evaluation expression (3) can be calculated in $O(M)$ operations.

5 Experiments

The goal of our experiments is to evaluate the inverted multi-index structure and, in particular, its applicability to the task of the approximate nearest neighbor search via the Multi-D-ADC system. Our experiments thus compare the performance of the inverted multi-index with the standard inverted index. We also compare different variants of the inverted multi-index, including the second- and the fourth-order multi-indices, the combination of the inverted multi-index with the PCA compression, and in addition evaluate the recent improvements of the Multi-D-ADC system suggested by us and by other researchers.

Below, we group the experiments according to the following three data processing tasks:

1) *Indexing.* Here, we compare different structures that can index large datasets of vectors. The structures are compared in their ability to return candidate lists with high recall in a small amount of time, when given a nearest neighbor query.
2) *Fast approximate nearest neighbor search.* Here we compare the performance of joint systems that include an indexing structure and a state-of-the-art reranking procedure (IVFADC, Multi-D-ADC, and the improved versions of Multi-D-ADC).
3) *Detection of near-duplicate images.* While the first two groups of experiments focus on the nearest-neighbor search w.r.t. the Euclidean metric, in the third group we evaluate the ability to retrieve near duplicates based on holistic GIST descriptors (which is correlated, but not identical to Euclidean nearest neighbor search).

Through the experiments we use the following datasets:

BIGANN: This dataset used in the majority of our experiments was introduced in [13] and contains 1 billion of 128-dimensional SIFT descriptors [32] extracted from natural images. The ground truth (true Euclidean nearest neighbors) for a hold-out set of 10,000 queries is provided with the dataset.

80 million Tiny Images: This dataset contains 384-dimensional GIST [33] descriptors corresponding to 80 million *Tiny Images* [9]. For this set, we picked a subset of 100 vectors and computed their Euclidean nearest neighbors within the rest of the dataset through exhaustive search thus obtaining the query set (which was excluded from the original dataset).

Augmented Copydays: The Copydays dataset was proposed in [34] for evaluating the robustness of image descriptors against artificial image transformations. The dataset contains 157 original images and their "copies" (results of JPEG, cropping and strong attacks). For each of original images we took four most similar images (10% and 20% crops and 75 and 50 JPEG quality factors) and considered them as ground truth duplicates of the original image. We calculated GIST-descriptors [33] of these $157 \times (1+4) = 785$ images and added to them 80 million Tiny images GISTs as distractors in order to emulate a large-scale near-duplicate search problem.

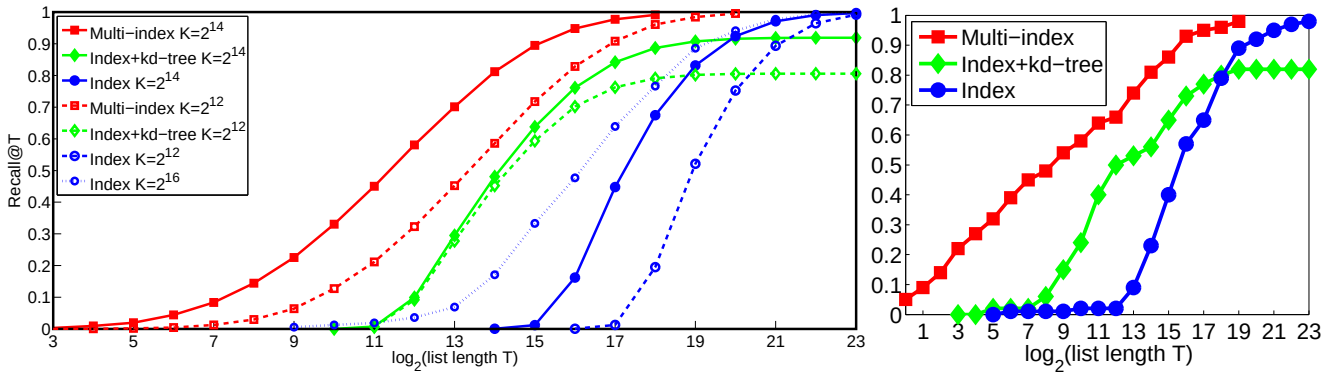
5.1 Indexing performance

In these experiments we study the quality of candidate lists produced by the multi-index. We report different measurements related to list lengths, timings, and the *recall*, which is defined as the probability of finding the *first* nearest neighbor of a query in a list returned by a certain system. This probability is always evaluated by averaging the rate of success (true nearest neighbor is on the list) over the available query set. In practice, the performance of retrieving other nearest neighbors (beyond the first one) is often important, however, this performance is highly correlated with the ability to retrieve the first nearest neighbor, and is therefore omitted from this evaluation. All timings were obtained on a single core of Intel Xeon 2.40 GHz CPU (using BLAS instructions in the single-thread mode).

5.1.1 Candidate list quality

To evaluate the quality of candidate lists we compare the recall of a second-order inverted multi-index and an inverted index for the same codebook size K. We perform this comparison for $K = 2^{14}$ for the BIGANN and 80 million dataset and, additionally, for a smaller $K = 2^{12}$ for the BIGANN dataset. For a set of predefined list lengths T (powers of two) and for each query, we traverse both data structures concatenating the lists stored in the entries. The traversal stops one step before the concatenated list length exceeds the predefined length T. Figure 4 plots the recall of such lists versus the length T (to which we refer as *recall@T*). In general, for a fixed K, the advantage of multi-indices over indices is very significant for the whole range of list lengths. For the SIFT1B dataset we also present a performance of an inverted index with significantly larger codebook $K = 2^{16}$. While for such K the performance of the inverted index improves considerably, it is still uniformly worse for all list lengths than the performance of the inverted multi-index for $K = 2^{12}$.

We then evaluate an additional baseline. As kd-trees [11] have emerged as a popular tool for working with very large codebooks, we took an even larger codebook (2^{18} codewords) and used a kd-tree (`vl_feat` [35] implementation)



1 billion SIFTs, $K = 2^{14}$ (solid), $K = 2^{12}$ (dashed), $K = 2^{16}$ (dotted) — 80 million GISTs, $K = 2^{14}$

Fig. 4: Recall as a function of the candidate list length. For the same codebook size K, we compare three systems with similar retrieval and construction complexities: an inverted index with K codewords, an inverted index with larger codebook (2^{18} codewords) sped up by a kd-tree search with a maximum of K comparisons, a second-order inverted multi-index with codebooks having K codewords. For billion-scale dataset we also provide results for an inverted index with $K = 2^{16}$ codewords which requires more runtime for quantization. In all experiments, multi-indices returned shorter lists with higher recall.

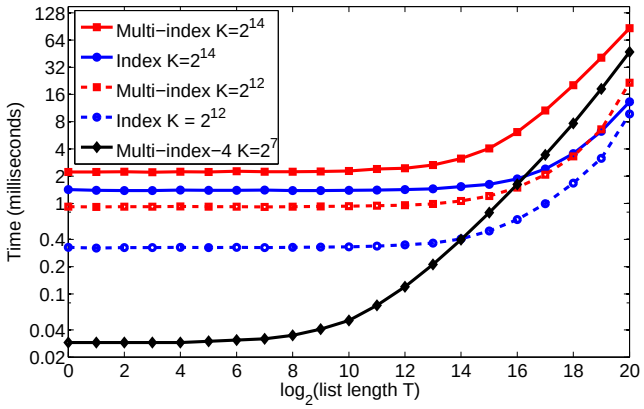


Fig. 5: Time (in milliseconds) required to retrieve a list of a particular length from the inverted multi-index and index on the BIGANN dataset.

to match the queries and the dataset vectors to this codebook (thus replacing the exhaustive search within the inverted index quantization with the fast approximate search). For a fair comparison, we limited the number of vector distance evaluations within the kd-tree to the respective K (either 2^{14} or 2^{12}). As can be seen in Figure 4, the new baseline is more competitive in the low recall area than the standard inverted index with the same values of K. This version, however, performs worse than the inverted index when high recall is needed. Overall, the recall@T of both baselines was uniformly worse than the recall@T of the inverted multi-indices in our experiments. Both, kd-trees and multi-indices incur some computational overhead over inverted indices (tree search and multi-sequence algorithm, respectively) and we now address the question how big this overhead is for the inverted multi-indices.

5.1.2 Retrieval speed

We give the timings for the second-order inverted multi-indices ($K = 2^{12}, K = 2^{14}$) on the BIGANN dataset as a function of the requested list length in Figure 5. The multi-index retrieval time essentially remains flat until the list length grows into many thousands, which means that the computational cost of the multi-sequence algorithm remains small compared to the quantization. We also give the timing curves for inverted indices with $K = 2^{12}, 2^{14}$. Their approximately two-fold speed advantage over the second-order indices (for the same K) stems most likely from the particular efficiency of vector instructions (BLAS library) on our CPU. This efficiency makes matching against codebooks faster in the inverted index case despite the same number of scalar operations.

Put together, Figure 4 and Figure 5 demonstrate the advantage of the second-order inverted multi-index over the standard inverted index. Thus, the multi-index with $K = 2^{12}$ provides much higher recall and is faster to query than the inverted index with $K = 2^{14}$ even when the BLAS instructions are used. In Figure 5, we also provide timings for the fourth-order multi-index and small K. Here, querying for short list lengths is much faster, however the overhead from the multi-sequence algorithm kicks in at shorter lengths (hundreds) exhibiting the main weakness of higher-order inverted multi-indices. We perform more comparisons involving the fourth-order multi-index below.

5.1.3 Multi-index + PCA

As discussed above, the computational bottleneck of the inverted multi-index is the computation of distances between the query vector and the codewords. The multi-index allows to reduce the size of vocabularies but the quantization still remains linear in the space dimensionality M. It is therefore natural desire to combine the multi-index with dimensionality reduction, e.g. with principal component analysis (PCA), which is the most popular dimensionality reduction method. Below, we describe the experiments conducted with the BIGANN dataset, the second-order multi-index with $K = 2^{14}$ and the four-fold dimensionality reduction that replaces 128-dimensional SIFTs with 32-dimensional

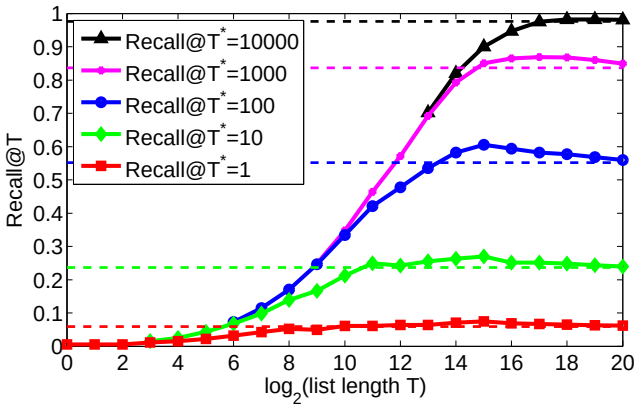


Fig. 7: Recall@T^* ($T^* = 1$ to 10000) of the Multi-ADC system (storing $m = 8$ extra bytes per vector for reranking) for the BIGANN dataset. The curves correspond to the Multi-ADC system that reranks a candidate list of a certain length T (x-axis) returned by the second-order multi-index ($K = 2^{14}$), while the flat dashed lines corresponds to the system that reranks the entire dataset. After reranking a tiny part of the billion-size dataset, Multi-ADC is able to match the performance of the exhaustive search-based system.

vectors. Our experiments with other output dimensionalities lead to similar results.

It turned out that there are two ways to combine PCA with the inverted multi-index, which lead to quite different efficiency of such combination:

Naive approach. This is the most obvious strategy. We first apply PCA to initial 128-dimensional vectors, truncating the top 32 principal components. To balance the variance, we multiply the resulting vectors by a random rotation matrix, and then build the multi-index on the resulting vectors.

PQ-aware approach. More efficient strategy performs PCA while taking the splitting of the dimensions into account. Thus, two independent PCA compressions are applied to 64-dimensional halves of initial vectors. The top 16 principal components from each half are retained. The resulting 16 dimensional vectors are multiplied by a random rotation matrix, and the inverted multi-index is built on the concatenations of the pairs of the resulting 16-dimensional vectors.

The indexing performance for both strategies are presented in Figure 6 in the form of recall@T curves. As one could expect the recall drop is smaller with the PQ-aware strategy because process of forming compressed vectors encourages independence between halves. In fact, the drop in the recall for the PCA-PQ-aware strategy is quite small, perhaps negligible for many applications, which means that PCA compression does not affect the quality of candidate lists returned by the second-order multi-index.

5.2 Approximate nearest neighbor search

The goal of these experiments is to evaluate the performance of the Multi-ADC and the Multi-D-ADC systems (built on top of the second-order inverted multi-index).

5.2.1 Multi-ADC

In the first experiment, we evaluate the Multi-ADC system with $m = 8$ extra bytes per vector (each vector is split into 8 dimension chunks and the PQ vocabularies of size 256 are used). Figure 7 then gives recall@T^* for $T^* = 1, 10, 100, 1000, 10000$ (different curves) on the BIGANN dataset as a function of the original candidate list length T returned by the inverted multi-index. As a baseline, we give the performance of the ADC system of [12] that essentially reranks the entire dataset ($T = 1$ billion), which takes several seconds per query. Figure 7 shows that, depending on T^*, it is sufficient to query only few hundred to few tens of thousand (i.e. a tiny fraction of the entire billion-size dataset) to match the performance of a system that reranks the entire dataset. At this point, the shortcomings of lossy compression within ADC seem to supersede (on average) whatever retrieval errors are made within the inverted multi-index. Curiously, the curves for Multi-ADC actually rise above the performance of full reranking before converging to it. We believe that this effect can have the following explanation. Because the PQ encoding is lossy, some "nasty" vectors are considered to be closer than the true nearest neighbor (NN) *after* reranking. In some cases, as T grows, the true nearest neighbor first enters the top T^* short list but then "sinks" out of it, as more and more of such "nasty" vectors enter the list of T candidate points.

5.2.2 Multi-D-ADC vs IVFADC

In this set of experiments, we compare the performance (recall@T^* and timings) of the Multi-D-ADC system for $T^* = 1, 10, 100$, $T = 10000, 30000, 100000$, and the number of extra bytes $m = 8, 16$. This performance is summarized in Table 1. For the *Tiny Images* dataset, we visualize few qualitative results of retrieval with Multi-D-ADC in Figure 8. For the BIGANN dataset, we give the recall and timings for our own re-implementation of the IVFADC system closely following the description in [12], [13]. We also reproduce the performance for the IVFADC system (state-of-the-art for $m = 8$ extra bytes) and for IVFADC+R system (state-of-the-art for $m = 16$ extra bytes) from [13] (the timings are thus computed on a different CPU).

Overall, it can be observed that for the same level of compression, the use of the inverted multi-indices gives Multi-D-ADC a substantial speed advantage over IVFADC(+R). This is achieved because Multi-D-ADC has to rerank much shorter candidate lists (tens of thousands vs. hundreds of thousand) to achieve similar or better recall values compared to IVFADC(+R). The memory overhead of Multi-D-ADC compared to IVFADC(+R) is about 8% (∼13GB vs. ∼12GB) for $m = 8$ and about 5% (∼21GB vs. ∼20GB) for $m = 16$ (all numbers include 4GB that are required to store point IDs).

Number of cells in IVFADC. As a baseline we choose IVFADC with $K = 2^{16}$ coarse cells. In general, larger codebooks would provide better short-lists (due to finer space partition) and more precise reranking (as the displacements encoded by the fine codebooks within Multi-D-ADC have smaller magnitudes). But the usage of larger codebooks in the IVFADC results in slower query quantization. Thus, the quantization of a 128-dimensional SIFT with $K = 2^{16}$ takes

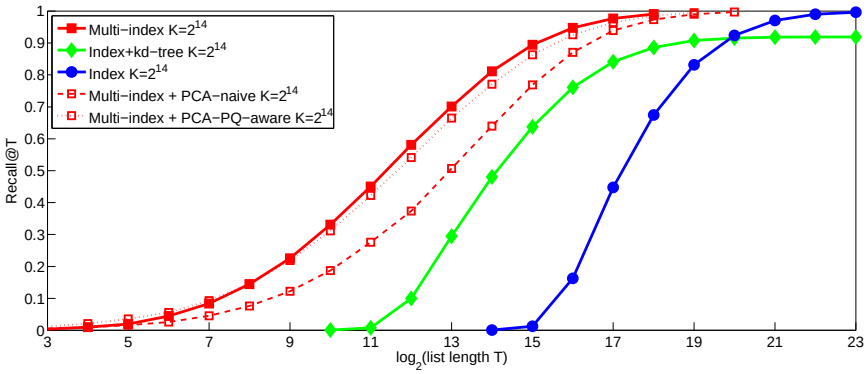


Fig. 6: Recall as a function of the candidate list length as in Figure 4 with added curves for different PCA strategies. Even with lossy PCA compression, the quality of candidate lists from multi-indices is higher than from the baseline systems. We also note that the PQ-aware strategy is significantly better than naive strategy to combine PCA and inverted multi-indices.

System	Number of cells	List len.T	R@1	R@10	R@100	Time(ms)	Memory(Gb)
BIGANN, 1 billion SIFTs, 8 bytes per vector							
IVFADC [13]	2^{13}	8 million	$0.112_{(0.088)}$	$0.343_{(0.372)}$	$0.728_{(0.733)}$	$155_{(74)}$	12
IVFADC [13]	2^{16}	600000	0.124	0.414	0.772	25	12
Multi-D-ADC	$2^{14} \times 2^{14}$	10000	0.153	0.473	0.707	2	13
Multi-D-ADC	$2^{14} \times 2^{14}$	30000	0.161	0.506	0.813	4	13
Multi-D-ADC	$2^{14} \times 2^{14}$	100000	0.162	0.515	0.854	11	13
BIGANN, 1 billion SIFTs, 16 bytes per vector							
IVFADC+R [13]	2^{13}	8 million	(0.262)	(0.701)	(0.962)	(116*)	20
IVFADC [13]	2^{16}	600000	0.311	0.750	0.923	28	20
Multi-D-ADC	$2^{14} \times 2^{14}$	10000	0.303	0.672	0.742	2	21
Multi-D-ADC	$2^{14} \times 2^{14}$	30000	0.325	0.762	0.883	5	21
Multi-D-ADC	$2^{14} \times 2^{14}$	100000	0.332	0.799	0.959	16	21
Tiny Images, 80 million GISTs, 8 bytes per vector							
Multi-D-ADC	$2^{14} \times 2^{14}$	10000	0.317	0.455	0.604	3	<1
Multi-D-ADC	$2^{14} \times 2^{14}$	30000	0.317	0.485	0.673	4	<1
Multi-D-ADC	$2^{14} \times 2^{14}$	100000	0.317	0.485	0.673	11	<1
Tiny Images, 80 million GISTs, 16 bytes per vector							
Multi-D-ADC	$2^{14} \times 2^{14}$	10000	0.317	0.544	0.653	3	<1
Multi-D-ADC	$2^{14} \times 2^{14}$	30000	0.326	0.574	0.733	5	<1
Multi-D-ADC	$2^{14} \times 2^{14}$	100000	0.327	0.584	0.852	17	<1

TABLE 1: The performance (recall for the top-1, top-10, and top-100 matches after reranking + time in milliseconds) of the Multi-D-ADC system (based on the second-order multi-index with K=2^{14}) for different datasets, different compression levels. We also give the performance of the IVFADC and IVFADC+R (our reimplementation for IVFADC as well as numbers reproduced from [13] in brackets – the timings are not directly comparable in the latter case).

about $3-4$ milliseconds in our experiments, and for the larger codebooks ($K > 2^{16}$) the quantization alone is slower then the overall time spent on quantization and reranking within Multi-D-ADC (see the typical timings in Table 1). While one possible solution could be to use approximate quantization, for example, via kd-trees, Figure 4 shows that for high levels of recall exact quantization with smaller codebooks should be preferred. For values of recall higher than 0.9 exact quantization with $K = 2^{16}$ provides better short-lists than approximate quantization with $K = 2^{18}$.

The second disadvantage of having very large codebook sizes K, is the amount of time spent on building the index. Since during the index construction, all time is spent on quantization and there is no reranking process involved, the time spent is almost linear in K (a sublinearity is introduced because of the BLAS instructions, however in our experiments, this sublinearity is rather small once K becomes large). Hence, building a multi-index of a large dataset is invariably faster than building a standard inverted index with a similar recall.

5.2.3 *Second-order vs Fourth-order multi-index.*

We also compare the performance of the second-order multi-index and the fourth-order multi-index with the same number of effective codewords. The results are summarized in Table 2. As one can see the *Multi-4-D-ADC* system (based on the fourth-order inverted multi-index) produces short list of candidates faster than the *Multi-D-ADC* system (based on the second-order inverted multi-index) but the recall is significantly lower. The reason is the effective codewords in the Multi-D-ADC and the Multi-4-D-ADC systems are produced under the assumption of independent halves and quarters of initial points respectively. Obviously the assumption of independent quarters is stronger. As a result, the effective codewords in the Multi-4-D-ADC system describe the structure of initial point space worse, which causes the drop in the recall. Moreover the timing advantage of the Multi-4-D-ADC is not large, which probably suggests that Multi-D-ADC system would be a preferred choice for nearest neighbor search in most cases. Below, we also com-

System	R@1	R@10	R@100	Time(ms)
BIGANN, 1 billion SIFTs, 8 bytes per vector				
Multi-D-ADC	0.153	0.473	0.707	2
Multi-4-D-ADC	0.093	0.276	0.407	1
BIGANN, 1 billion SIFTs, 16 bytes per vector				
Multi-D-ADC	0.303	0.672	0.742	2
Multi-4-D-ADC	0.191	0.391	0.427	1
Tiny Images, 80 million GISTs, 8 bytes per vector				
Multi-D-ADC	0.317	0.455	0.604	3
Multi-4-D-ADC	0.247	0.426	0.495	1
Tiny Images, 80 million GISTs, 16 bytes per vector				
Multi-D-ADC	0.317	0.544	0.653	3
Multi-4-D-ADC	0.307	0.475	0.523	1

TABLE 2: The performance of the Multi-D-ADC and Multi-4-D-ADC systems with the same number of effective codewords (the second-order multi-index with $K{=}2^{14}$ and the fourth-order multi-index with $K{=}2^{7}$ are used for indexing) for different datasets. In all experiments Multi-4-D-ADC system produces shortlist of given length (10,000) faster because of fast quantization. But the quality of the Multi-4-D-ADC shortlist is considerably worse than the quality of the Multi-D-ADC shortlist. This difference translates into the drop of the recall after the reranking is performed.

System	R@1	R@10	R@100	Time(ms)
BIGANN, 1 billion SIFTs, 8 bytes per vector				
Original points	0.153	0.473	0.707	2
PCA-naive	0.116	0.348	0.525	1
PCA-PQ-aware	0.116	0.387	0.645	1

TABLE 3: The performance of different strategies to combine the second-order multi-index and the PCA-based four-fold compression. The speedup is the same for both strategies but the recall of the PQ-aware strategy is higher, since this strategy encourages independence between subvectors thus improving the perfomance of the multi-index.

pare the second and the fourth-order multi-indices for the near-duplicate detection

5.2.4 Multi-D-ADC + PCA

In this set of experiments we combine the Multi-D-ADC system and PCA, while using the naive and the PQ-aware strategies (see Section 5.1.3). As before, we use the BIGANN dataset and consider four-fold PCA dimensionality reduction to 32 dimensions. In Section 5.1.3 lossy PCA compression affected only indexing quality. In this experiment, the PCA compression also affects the reranking quality, as the additional bytes within Multi-D-ADC encode lossy-compressed displacement vectors. The results for both PCA strategies are presented in Table 3. As one could expect, the speedup is the same for both approaches. Figure 6 suggests that the PCA compression does not affect indexing quality by much in the PQ-aware case, which means that closest "coarse" cells of multi-index can be found based only on main principal components of a query. Unfortunately, Table 3 shows a serious drop in the recall after reranking even for the PQ-aware strategy. This suggests that minor components truncated in the PCA compression are necessary for the accurate "fine reranking".

5.2.5 Improved versions of Multi-D-ADC

Since the initial publication [26] two improvements for the Multi-D-ADC system have been suggested. First,

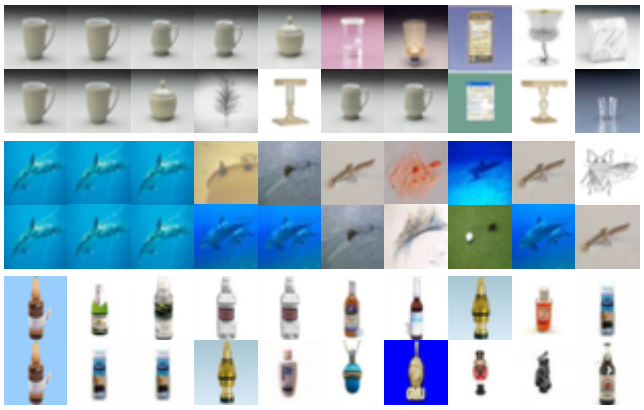


Fig. 8: Retrieval examples on the Tiny Images dataset (the images associated with GIST vectors are shown). In each of the three row pairs, the left-most images correspond to the query, the top row corresponds to Euclidean nearest neighbors found by exhaustive search, the bottom row are the top matches returned by the Multi-D-ADC system ($K = 2^{14}$, $m = 16$ extra bytes). Empirically, for most examples, we observed that the top matches returned by a Multi-D-ADC are similar in terms of semantic similarity to the exhaustive search on uncompressed vectors (top two rows) with few exceptions (bottom row).

Ge et al. [27] have improved the accuracy of the system by replacing the product quantization with the optimized product quantization (OPQ) [28], [29] for both indexing and compression (i.e. both at the coarse level and at the fine level). In a nutshell, OPQ is an extension of PQ which finds a data-specific orthogonal transformation which makes product subspaces less correlated. For some kinds of data a preprocessing with such a transformation results in significantly higher performance. The resulting ANN search system (OMulti-D-OADC) improves the accuracy of Multi-D-ADC considerably.

Secondly, in Kalantidis et al. [30] and independently in our report [31], it was suggested that in addition to using the optimized product quantization, a separate i.e. *local* second-level (fine) codebook can be learned for each coarse-level codeword in order to encode the displacements of vectors that belong to multi-index cells that share this codeword. Here, we refer to this system as OMulti-D-OADC-Local.

Table 4 demonstrates the recall levels achieved by OMulti-D-OADC and OMulti-D-OADC-Local systems for the billion-scale SIFT1B dataset. The OMulti-D-OADC-Local system achieves significantly higher recall and provides current state-of-the-art for this dataset. While the improvement in accuracy from the use of the local codebooks is substantial, we notice that the use of local codebooks precludes the speed optimization we discuss in Section 4, which results in increased runtimes. Also, the space required for storing local codebooks might be considerable (2 Gb in our settings). The IVFADC system also allows to use the OPQ at the fine level for compression. We refer to this modification as IVFOADC and present its performance in Table 4 for comparison.

Importantly, the gap between IVFADC and Multi-D-ADC remains about the same, once rotation optimization is brought in. The use of local codebooks within OMulti-D-OADC-Local further increases this gap (as one would expect). It is certainly possible to use local codebooks with

System	Number of cells	l	R@1	R@10	R@100	Time(ms)	Memory(Gb)
BIGANN, 1 billion SIFTs, 8 bytes per vector							
IVFOADC	2^{16}	600000	0.138	0.451	0.810	25	12
OMulti-D-OADC [27]	$2^{14} \times 2^{14}$	10000	0.180	0.518	0.747	2	13
OMulti-D-OADC [27]	$2^{14} \times 2^{14}$	30000	0.184	0.548	0.848	4	13
OMulti-D-OADC [27]	$2^{14} \times 2^{14}$	100000	0.186	0.559	0.892	11	13
OMulti-D-OADC-Local [30], [31]	$2^{14} \times 2^{14}$	10000	0.268	0.644	0.776	6	15
OMulti-D-OADC-Local [30], [31]	$2^{14} \times 2^{14}$	30000	0.280	0.704	0.894	16	15
OMulti-D-OADC-Local [30], [31]	$2^{14} \times 2^{14}$	100000	0.286	0.729	0.952	50	15
BIGANN, 1 billion SIFTs, 16 bytes per vector							
IVFOADC	2^{16}	600000	0.315	0.764	0.955	28	20
OMulti-D-OADC [27]	$2^{14} \times 2^{14}$	10000	0.339	0.704	0.769	2	21
OMulti-D-OADC [27]	$2^{14} \times 2^{14}$	30000	0.360	0.792	0.901	5	21
OMulti-D-OADC [27]	$2^{14} \times 2^{14}$	100000	0.367	0.834	0.969	16	21
OMulti-D-OADC-Local [30], [31]	$2^{14} \times 2^{14}$	10000	0.421	0.755	0.782	7	23
OMulti-D-OADC-Local [30], [31]	$2^{14} \times 2^{14}$	30000	0.454	0.862	0.908	19	23
OMulti-D-OADC-Local [30], [31]	$2^{14} \times 2^{14}$	100000	0.467	0.914	0.976	66	23

TABLE 4: The results for (our implementations of) the improved versions of the Multi-D-ADC system. OMulti-D-OADC proposed by Ge et al. [27] replaces product quantization with the optimized product quantization. OMulti-D-OADC-Local system proposed in [30], [31] further increases the accuracy (at the cost of extra time and memory) through the use of local codebooks in each coarse cell. The IVFOADC is a modification of the IVFADC which uses the OPQ for database points compression.

IVFOADC, however the memory required to store local codebooks would grow significantly (e.g. 8 Gb for $K = 2^{16}$).

5.3 Near-duplicate detection

We also apply the inverted multi-index to the problem of large-scale duplicate detection [36], [37], [38]. In these experiments we used the GIST-descriptors of Copydays [34] merged with the GIST descriptors of 80 million *Tiny Images* [9] thus having a dataset with known subsets of near duplicates. We built the second-order ($K = 2^{14}$) and the fourth-order ($K = 2^{7}$) multi-indices on this dataset and use descriptors of the original Copydays images as queries to these multi-indices. The main measure of near-duplicate detection quality we take the *recall@T*, which in this case means the percentage of groundtruth near-duplicates in the candidate lists of length T returned by a multi-index, averaged over all queries. The results presented on Figure 9 and Figure 10 are consistent with the relative performance of the two indices for nearest neighbor search (Table 2).

Thus, as shown in Figure 10, the fourth-order system produces a candidate list with a given recall faster. However, as can be seen from Figure 9 for any recall level, candidate lists produced by the second order multi-index is significantly shorter. Overall, as in most systems the returned candidate lists are likely to be reranked/verified in a certain way (e.g. by comparing some hash values between the query image and the indexed images), the system based on the second-order multi-index is likely to be faster for any certain level of desired recall (after the verification/reranking time is factored in).

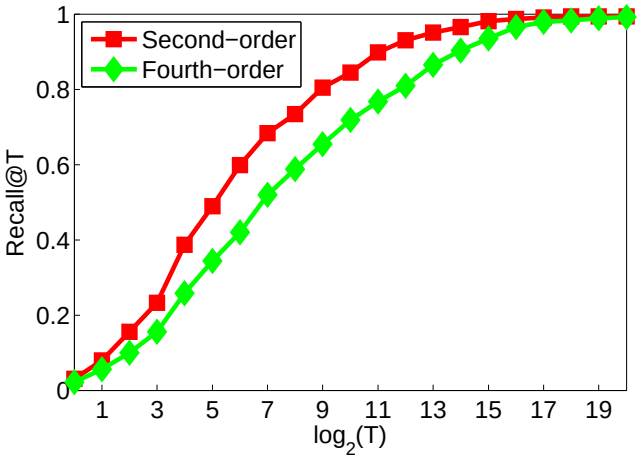


Fig. 9: Near-duplicate retrieval recall@T of the second- and the fourth-order multi-indices as a function of the candidate list length. For all reasonable lengths the second-order multi-index produces lists with higher recall.

6 Discussion

We have introduced the *inverted multi-index*, which is a generalization of a standard inverted index for the large-scale retrieval in the datasets of high-dimensional vectors. In our evaluation, second-order multi-indices significantly outperformed standard inverted indices in terms of the accuracy of the returned candidate lists given the same runtime budget. This advantage over the inverted indices, also translates to the complete systems for approximate nearest neighbor search that combine candidate list generation with reranking. Here, the fact that the inverted multi-index can generate much shorter candidate lists to achieve the same level of recall, allows to query billion-scale datasets in a few seconds.

We have also compared the second-order and the fourth-order inverted multi-indices for the tasks of nearest neighbor search and duplicate detection and found out that the second order multi-index would be a better choice in most situations. Since the standard inverted index can be regarded as a first-order inverted multi-index, we believe that the second order might be a sweet spot. One important consideration that however may force to prefer the fourth-order multi-index over the second-order multi-index is the

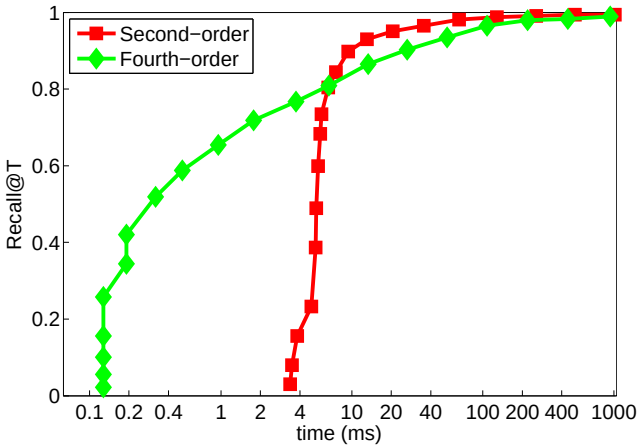


Fig. 10: Near-duplicate retrieval recall@T of the second- and the fourth-order multi-indices as a function of time. Because of faster quantization the fourth-order multi-index traverses the cells closest to queries and attains moderate recall values faster. For high recall values the second-order multi-index is preferable: it gains high recall faster because its traversal procedure better concentrates on the parts of space around the query (cf. Figure 9).

index construction time (which for simplicity was omitted from our experimental evaluation). Indeed, the multi-index construction is likely to have a time bottleneck at the quantization stage (assuming that the extra information such as PQ compression is fast to compute). As the quantization time would be dramatically (at least an order of magnitude) smaller for the fourth-order multi-index, it can be preferred whenever index construction time is important.

We have also looked into the combination of the inverted multi-index and the PCA dimensionality reduction, which allows to speedup search without significant drop in the recall of the returned candidate lists, as long as PCA is applied in a smart way that takes into account the split into dimension group.

Similarly to other works that rely on the product quantization, the efficiency of the inverted multi-index increases as the correlation between the dimension groups that the vectors are split into decreases. As shown above, the correlation between the halves of the SIFT or GIST vectors is low enough for the inverted multi-index to perform well. Generally, the input vectors can be transformed by the transform that decreases the correlation between the parts of the vector. Finding such transformation was the subject of the recent work of Ge et al. [28], and their experiments suggest that the performance of the inverted multi-indices can be improved as a result of such transformation. Thus, they were able to improve our results on the BIGANN dataset through the use of such transformation (while leaving the use of the multi-index unchanged otherwise). Here, we evaluate the suggested improvement along with the idea of local codebooks [30], [31] and confirm that they can substantially improve the accuracy of the Multi-D-ADC system built on top of the inverted multi-index.

Apart from their use within the nearest neighbor search or duplicate detection systems, multi-indices can be also used within retrieval systems that combine the candidate lists returned for multiple descriptors extracted from the same query image [1]. There, replacing candidate lists corresponding to a single codeword with something closer to nearest neighbor search has been shown to improve the accuracy significantly albeit at a considerable computational cost (c.f. [39], [40]). Furthermore, it is straightforward to replace the (square of the) Euclidean distance within the multi-index with any other additive distance measure or kernel; it will thus be interesting to evaluate inverted multi-indices within large-scale machine learning systems, in particular those utilizing exemplar SVMs [41].

Acknowledgments

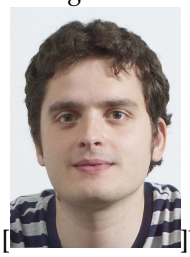
We would like to thank two CVPR'12 reviewers for their exceptionally useful reviews.

References

[1] J. Sivic and A. Zisserman, "Video Google: A text retrieval approach to object matching in videos," in *ICCV*, 2003.
[2] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman, "Object retrieval with large vocabularies and fast spatial matching," in *CVPR*, 2007.
[3] O. Boiman, E. Shechtman, and M. Irani, "In defense of nearest-neighbor based image classification," in *CVPR*, 2008.
[4] J. Deng, A. C. Berg, K. Li, and F.-F. Li, "What does classifying more than 10, 000 image categories tell us?" in *ECCV*, 2010.
[5] T. Malisiewicz and A. A. Efros, "Beyond categories: The visual memex model for reasoning about object relationships," in *NIPS*, 2009.
[6] J. Hays and A. A. Efros, "Scene completion using millions of photographs," *ACM Trans. Graph.*, vol. 26, no. 3, 2007.
[7] "Google Goggles," `http://www.google.com/mobile/goggles`.
[8] H. Lejsek, B. T. Jónsson, and L. Amsaleg, "NV-Tree: nearest neighbors at the billion scale," in *ICMR*, 2011.
[9] A. Torralba, R. Fergus, and W. T. Freeman, "80 million tiny images: A large data set for nonparametric object and scene recognition," *TPAMI*, vol. 30, no. 11, 2008.
[10] D. Nistér and H. Stewénius, "Scalable recognition with a vocabulary tree," in *CVPR*, 2006.
[11] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, 1975.
[12] H. Jégou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *TPAMI*, vol. 33, no. 1, 2011.
[13] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg, "Searching in one billion vectors: Re-rank with source coding," in *ICASSP*, 2011.
[14] C. D. Manning, P. Raghavan, and H. Schutze, *Introduction to Information Retrieval*, C. U. Press, Ed., 2008.
[15] O. Chum, J. Philbin, J. Sivic, M. Isard, and A. Zisserman, "Total recall: Automatic query expansion with a generative feature model for object retrieval," in *ICCV*, 2007.
[16] J. Philbin, M. Isard, J. Sivic, and A. Zisserman, "Descriptor learning for efficient retrieval," in *ECCV*, 2010.
[17] W.-L. Zhao, X. Wu, and C.-W. Ngo, "On the annotation of web videos by efficient near-duplicate search," in *Transactions on Multimedia*, 2010.
[18] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *STOC*, 1998.
[19] M. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the Symposium on Theory of Computing*, 2002.
[20] Q. Lv, M. Charikar, and K. Li, "Image similarity search with compact data structures," in *CIKM*, 2004.
[21] R. Salakhutdinov and G. E. Hinton, "Semantic hashing," *Int. J. Approx. Reasoning*, vol. 50, no. 7, 2009.
[22] A. Torralba, R. Fergus, and Y. Weiss, "Small codes and large image databases for recognition," in *CVPR*, 2008.
[23] M. Raginsky and S. Lazebnik, "Locality-sensitive binary codes from shift-invariant kernels," in *NIPS*, 2009.
[24] H. Jegou, M. Douze, C. Schmid, and P. Pérez, "Aggregating local descriptors into a compact image representation," in *CVPR*, 2010.

[25] H. Jegou, M. Douze, and C. Schmid, "Hamming embedding and weak geometric consistency for large scale image search," in *ECCV*, 2008.
[26] A. Babenko and V. Lempitsky, "The inverted multi-index," in *CVPR*, 2012.
[27] T. Ge, K. He, Q. Ke, and J. Sun, "Optimized product quantization," Tech. Rep., 2013.
[28] ——, "Optimized product quantization," MSR-TR-2013-59, Tech. Rep., 2013.
[29] M. Norouzi and D. J. Fleet, "Cartesian k-means," in *CVPR*, 2013.
[30] Y. Kalantidis and Y. Avrithis, "Locally optimized product quantization for approximate nearest neighbor search," in *in Proceedings of International Conference on Computer Vision and Pattern Recognition (CVPR 2014)*. IEEE, 2014.
[31] A. Babenko and V. Lempitsky, "Improving bilayer product quantization for billion-scale approximate nearest neighbors in high dimensions," *CoRR*, vol. abs/1404.1831, 2014.
[32] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *IJCV*, vol. 60, no. 2, 2004.
[33] A. Oliva and A. Torralba, "Modeling the shape of the scene: A holistic representation of the spatial envelope," *IJCV*, vol. 42, no. 3, 2001.
[34] M. Douze, H. Jegou, H. Sandhawalia, L. Amsaleg, and C. Schmid, "Evaluation of gist descriptors for web-scale image search," in *CIVR*, 2009.
[35] A. Vedaldi and B. Fulkerson, "VLFeat: An open and portable library of computer vision algorithms," `http://www.vlfeat.org/`.
[36] M. I. David C. Lee, Qifa Ke, "Partition min-hash for partial duplicate image discovery," in *ECCV*, 2010.
[37] O. Chum, J. Philbin, M. Isard, and A. Zisserman, "Scalable near identical image and shot detection," in *CIVR*, 2007.
[38] Y. Ke, R. Sukthankar, and L. Huston, "Efficient near-duplicate detection and sub-image retrieval," in *ACM Multimedia*, 2004.
[39] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman, "Lost in quantization: Improving particular object retrieval in large scale image databases," in *CVPR*, 2008.
[40] H. Jégou, M. Douze, and C. Schmid, "Exploiting descriptor distances for precise image search," Research Report, 2011.
[41] T. Malisiewicz, A. Gupta, and A. A. Efros, "Ensemble of exemplar-svms for object detection and beyond," in *ICCV*, 2011, pp. 89–96.

[]Artem Babenko received his MS degree in computer science from Moscow Institute of Physics and Technology (MIPT) in 2012. Currently, he is a researcher at Yandex and also holds a teacher assistant position in National Research University Higher School of Economics (HSE). Artem's research is focused on problems of large scale image retrieval and recognition.



[]Victor Lempitsky is an assistant professor at Skolkovo Institute of Science and Technology (Skoltech), which is a new research university in Moscow area. Prior to that, he hold a researcher position at Yandex, and postdoctoral positions with the Visual Geometry Group of Oxford University, as well as with the Computer Vision group at Microsoft Research Cambridge. Victor has a PhD ("kandidat nauk") from Moscow State University (2007). His research interests are in various aspects of computer vision such as visual recognition, image understanding, fine-grained classification, visual search, and biomedical image analysis.