Reconfigurable Inverted Index

Yusuke Matsui
National Institute of Informatics
matsui@nii.ac.jp

Ryota Hinami
The University of Tokyo
hinami@nii.ac.jp

Shin'ichi Satoh
National Institute of Informatics
satoh@nii.ac.jp

ABSTRACT

Existing approximate nearest neighbor search systems suffer from two fundamental problems that are of practical importance but have not received sufficient attention from the research community. First, although existing systems perform well for the whole database, it is difficult to run a search over a subset of the database. Second, there has been no discussion concerning the performance decrement after many items have been newly added to a system. We develop a reconfigurable inverted index (Rii) to resolve these two issues. Based on the standard IVFADC system, we design a data layout such that items are stored linearly. This enables us to efficiently run a subset search by switching the search method to a linear PQ scan if the size of a subset is small. Owing to the linear layout, the data structure can be dynamically adjusted after new items are added, maintaining the fast speed of the system. Extensive comparisons show that Rii achieves a comparable performance with state-of-the art systems such as Faiss.

CCS CONCEPTS

• **Information systems** → *Nearest-neighbor search*; *Search engine indexing*; *Multimedia and multimodal retrieval*; • **Computing methodologies** → *Visual content-based indexing and retrieval*;

KEYWORDS

Approximate nearest neighbor search; inverted index; product quantization; subset search; reconfigure

ACM Reference Format:
Yusuke Matsui, Ryota Hinami, and Shin'ichi Satoh. 2018. Reconfigurable Inverted Index. In *2018 ACM Multimedia Conference (MM '18), October 22–26, 2018, Seoul, Republic of Korea.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3240508.3240630

1 INTRODUCTION

In recent years, the approximate nearest neighbor search (ANN) has received increasing attention from various research communities [19]. Typical ANN systems operate in two stages. In the offline phase, database vectors are stored in the ANN system. These vectors may be converted to other forms, such as compact codes, for fast searching and efficient memory usage. In the online querying phase, the system receives a query vector. Similar items to the query are retrieved from the stored database vectors. Their identifiers (and optionally their distances to the query) are then returned. To handle large datasets, this search should be not only fast and accurate, but also memory efficient.

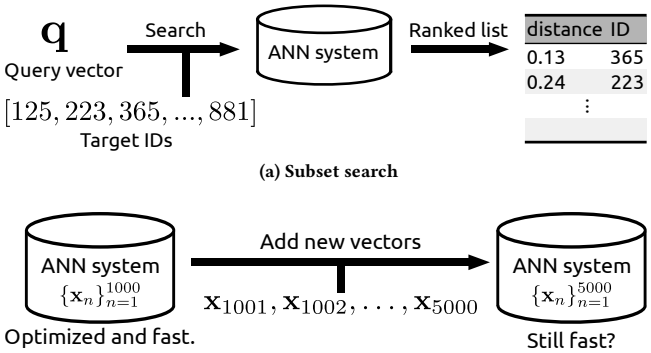

Figure 1: The two problems tackled in this paper. (a) The search is operated for a subset of a database, which is specified by the target identifiers. The search result (ranked list) should contain the specified items only. (b) Given a fast (optimized) ANN system, new vectors are added. Is the updated ANN system still fast?

Although many ANN methods have already been proposed, there are two critical problems of practical importance that have not received sufficient attention from the research community (Fig. 1).

- *Subset search* (Fig. 1a): Once database vectors are stored, modern ANN systems can run a search efficiently for the **whole** database. Surprisingly, however, almost no systems can run a search over a **subset** of the database[1]. For example, let us consider an image search problem, where the search is formulated as an ANN search over feature vectors. We assume that each image also has a corresponding shooting date. Given a query image, an ANN system can easily find similar images from the whole dataset. However, it is not trivial to find similar images that were taken on a target date (say, May 28 1987). Here, the search should not be conducted over the whole dataset, but rather over a subset of the dataset, where the subset is specified by identifiers of target images. The straightforward solution is to run the search and check whether or not the results were taken on May 28, but this post-checking can be drastically slow, especially if the size of the subset is small. Current ANN systems cannot provide a clear solution to this problem.
- *Performance degradation via data addition* (Fig. 1b): So far, the manner in which the search performance degrades when items are newly added has not been discussed. The number of database items is typically assumed to be provided when an ANN

[1] For example, the state-of-the-art systems Faiss [25] and Annoy [11] do not provide this functionality. See discussion at https://github.com/facebookresearch/faiss/issues/322, https://github.com/spotify/annoy/issues/263

MM '18, October 22–26, 2018, Seoul, Republic of Korea

ACM ISBN 978-1-4503-5665-7/18/10…$15.00
https://doi.org/10.1145/3240508.3240630

system is built. Parameters of the system are usually optimized by taking this number into consideration. However, in a practical scenario, new items might be often added to the system. Although the performance does not change while the number of new items is small, we can ask whether the system remains efficient even after 100× items are newly added. To put this another way, suppose that one would like to develop a search system that can handle 1,000,000 vectors in the future, but only has 1,000 vectors in the initial stage. In such a case, is the search fast even for 1,000 vectors?

We develop an ANN system that solves the above two problems, namely *reconfigurable inverted index (Rii)*. The key idea is extremely simple: storing the data **linearly**. Based on the well-known inverted file with product quantization (PQ) approach (IVFADC) [26], we design the data layout such that an item can be fetched by its identifier with a cost of $O(1)$. This simple but critical modification enables us to search over a subset of the dataset efficiently by switching to a linear PQ scan if the size of the subset is small. Owing to this linear layout, the granularity of a coarse assignment step can easily be controlled by running clustering again over the dataset whenever the user wishes. This means that the data structure can be adjusted dynamically after new items are added.

An extensive comparison with state-of-the-art systems, such as Faiss [25], Annoy [11], Falconn [41], and NMSLIB [39], shows that Rii achieves a comparable performance. For subset searches and data-addition problems for which the existing approaches do not perform well, we demonstrate that Rii remains fast in all cases.

Our contributions are summarized as follows.

- Rii enables efficient searching over a subset of the whole database, regardless of the size of the subset.
- Rii remains fast, even after many new items are added, because the data structure is dynamically adjusted for the current number of database items.

2 RELATED WORK

We review existing work that is closely related to our approach.

Locality-sensitive-hashing. Locality-sensitive-hashing (LSH) [15] can be considered as one of the most popular branches of ANN. Hash functions are designed such that the probability of collision is higher for close points than for points that are widely separated. Using these functions with hash tables, nearest items can be found efficiently. Although it has been said that LSH requires a lot of memory and is not accurate compared to data-dependent methods, a recent well-tuned library (FALCONN [1, 41]) using multi-probe technology [31] has achieved a reasonable performance.

Projection/tree-based approach. Space partitioning using a projection or tree constitutes another significant branch of ANN. Especially in the computer vision community, one of the most widely employed methods is FLANN [38]. Recently, the random projection forest-based method Annoy [11] achieved a good performance for million-scale data.

Graph traversal. Benchmark scores [4, 12] show that graph traversal-based methods [32, 33] achieve the current best performance (the fastest with a fixed recall) when the number of database items is around one million. These methods first create a graph where each node corresponds to a database item, which is called a navigable small world. Given a query, the algorithm starts from a random initial node. The graph is traversed to the node that is the closest to the query. In particular, the hierarchical version HNSW [33] with the highly optimized implementation NMSLIB [14] represents the current state-of-the-art. The drawback is that it tends to consume memory space, with a long runtime for building the data structure.

Product quantization. Product quantization (PQ) [26] and its extensions [5, 6, 8, 16, 18, 21, 24, 34, 40, 44, 48, 49] are popular approaches to handling large-scale data. Our proposed Rii method also follows this line. PQ-based methods compress vectors into short memory-efficient codes. The Euclidean distance between an original vector and compressed code can be efficiently approximated using a lookup table. Current billion-scale search systems are usually based on PQ methods, especially combined with an inverted index-based architecture [7, 20, 23, 29, 37, 42, 46]. Hardware-based acceleration has also recently been discussed [2, 3, 13, 28, 30, 45, 47]. An efficient implementation proposed by the original authors is Faiss [25, 28]. An extensive survey is given in [36].

3 BACKGROUND: PRODUCT QUANTIZATION

In this section, we will review product quantization (PQ) [26]. PQ compresses vectors into memory efficient short codes. The squared Euclidean distance between an input vector and the compressed code can be approximated efficiently. Owing to its memory-efficient form, PQ played a central role in large-scale ANN systems.

We first describe how to encode a vector. A D-dimensional input vector $\mathbf{x} \in \mathbb{R}^D$ is split into M sub-vectors. Each D/M-dimensional sub-vector is compared to Z pre-trained code words, and the identifier (an integer in $\{1, 2, \ldots, Z\}$) of the closest one is recorded. Using this, $\mathbf{x}$ is encoded as $\bar{\mathbf{x}}$, which is a tuple of M integers:

$$\mathbf{x} \mapsto \bar{\mathbf{x}} = [\bar{x}^1, \ldots, \bar{x}^M]^\top \in \{1, \ldots, Z\}^M, \tag{1}$$

where the mth sub-vector in $\mathbf{x}$ is quantized into $\bar{x}^m$. We refer to $\bar{\mathbf{x}}$ as a PQ-code for $\mathbf{x}$. Note that $\bar{\mathbf{x}}$ is represented by $M \log_2 Z$ bits, and we set Z to 256 in order to represent each code using M bytes.

Next, we show how to search over the PQ-codes given a query vector $\mathbf{q} \in \mathbb{R}^D$. First, a distance table $\mathbf{A} \in \mathbb{R}^{M \times Z}$ is computed online by comparing the query to the code words. Here, $A(m, z)$ is the squared Euclidean distance between the mth part of $\mathbf{q}$ and zth code word from the mth codebook. The squared Euclidean distance between the query $\mathbf{q}$ and the database vector $\mathbf{x}$ can be approximately computed using the PQ-code $\bar{\mathbf{x}}$, as follows:

$$d(\mathbf{q}, \mathbf{x})^2 \sim d_A(\mathbf{q}, \bar{\mathbf{x}})^2 = \sum_{m=1}^{M} A(m, \bar{x}^m). \tag{2}$$

This is called an asymmetric distance computation (ADC) [26], and can be performed efficiently, because only M fetches are required on $\mathbf{A}$. A search over N PQ-codes requires $O(DZ + MN)$.

4 RECONFIGURABLE INVERTED INDEX

Now, we introduce our proposed approach: reconfigurable inverted index (Rii). Let us define a query vector $\mathbf{q} \in \mathbb{R}^D$, N database vectors $\mathcal{X} = \{\mathbf{x}_n \in \mathbb{R}^D\}_{n=1}^N$, and target identifiers $\mathcal{S} \subseteq \{1, \ldots, N\}$. The subset-search problem is defined to find the R similar items to the

query from the subset of $\mathcal{X}$ specified by $\mathcal{S}$:

$$R\text{-}\operatorname*{argmin}_{s\in\mathcal{S}} \|\mathbf{q}-\mathbf{x}_s\|_2^2, \tag{3}$$

where the R-argmin operator finds the R arguments for which an objective function attains R (sorted) smallest values. The exact solution can be obtained by a time-consuming direct linear scan. Our goal is to approximately find nearest items in a fast and memory-efficient manner. Note that the problem turns out to be a usual ANN search if the whole database is set as the subset: $\mathcal{S} = \{1, \dots, N\}$.

4.1 Data Structure

First, N input database vectors $\mathcal{X}$ are encoded as PQ-codes $\bar{\mathcal{X}} = \{\bar{\mathbf{x}}_n\}_{n=1}^N$, where each $\bar{\mathbf{x}}_n \in \{1, \dots, Z\}^M$. These PQ-codes are stored **linearly**, meaning that they are stored in a single long array. Given an identifier n, fetching $\bar{\mathbf{x}}_n$ requires a computational cost of $O(1)$.

The PQ-codes are clustered into K groups for inverted indexing. First, K coarse centers $\bar{\mathcal{C}} = \{\bar{\mathbf{c}}_k\}_{k=1}^K$ are created by running the clustering algorithm [35] on $\bar{\mathcal{X}}$ (or its subset). Note that each coarse center is also a PQ-code $\bar{\mathbf{c}}_k \in \{1, \dots, Z\}^M$. Using these coarse centers, the database PQ-codes $\bar{\mathcal{X}}$ are clustered into K groups. The resulting assignments are stored as posting lists $\mathcal{W} = \{\mathcal{W}_k\}_{k=1}^K$, where each $\mathcal{W}_k$ is a set of identifiers of the database vectors whose nearest coarse center is the kth one:

$$\mathcal{W}_k = \{n \in \{1, \dots, N\} | a(n) = k\}. \tag{4}$$

Note that $a(n)$ is an assignment function, that is defined as $a(n) = \operatorname{argmin}_{k\in\{1,\dots,K\}} d_S(\bar{\mathbf{x}}_n, \bar{\mathbf{c}}_k)$, where d_S is a symmetric distance function that measures the distance between two PQ-codes [26, 35]. Finally, we store $\bar{\mathcal{X}}$, $\bar{\mathcal{C}}$, and $\mathcal{W}$ as a data structure for Rii. The total theoretical memory usage is $(N + K)M \log_2 Z + 32N$ bits if an integer is represented by 32 bits. We will show in Sec. 5.5 that this theoretical value is almost the same as the measured value.

Note that in a typical implementation of the original IVFADC [26] system, PQ-codes are stored in posting lists (not a single array). That is, $\{\bar{\mathbf{x}}_n | a(n) = k\}$ are chunked for each k and then stored. This would enhance the locality of the data, and improve the cache efficiency when traversing a posting list. However, the experimental results (Sec. 5.5) showed that this difference is not serious.

4.2 Search

We explain how to search for similar vectors using the data structure explained above. Our system provides two search methods: *PQ-linear-scan* and *inverted-index*. The former is fast when the size of a target subset is small, and the latter is fast when the size is large. Depending on the size, the faster method is automatically selected.

A search over a subset of a database is defined as a search on target PQ-codes denoted by the target identifiers $\mathcal{S} \subseteq \{1, \dots, N\}$. Note that we assume that the elements of $\mathcal{S}$ are **sorted**[2]. This is a slightly strong but reasonable assumption. Because $\mathcal{S}$ is sorted, it can be checked whether or not an item is contained in a set ($n \in \mathcal{S}$) with a cost of $O(\log_2 |\mathcal{S}|)$ using a binary search, where $|\mathcal{S}|$ is the number of elements in $\mathcal{S}$. Note again that a search over the whole dataset is available by setting $\mathcal{S} = \{1, \dots, N\}$.

[2] A set is denoted by calligraphic font, such as $\mathcal{X}$, and implemented by a single array.

Algorithm 1: PQLinearScan

Input: $\mathbf{q} \in \mathbb{R}^D$, # Query
$\bar{\mathcal{X}} = \{\bar{\mathbf{x}}_n\}_{n=1}^N$, # Database PQ-codes
$R \in \{1, \dots, N\}$, # # of returned items
$\mathcal{S} \subseteq \{1, \dots, N\}$ # Target subset identifiers
Output: $\mathcal{U} = \{\mathbf{u}_r\}_{r=1}^R$ s.t. $\mathbf{u}_r = [n_r, d_r] \in \{1, \dots, N\} \times \mathbb{R}$
n_r: r-th identifier. d_r: r-th distance.

```
1 A ← CompareCodewords(q) # Distance table
2 U ← ∅ # Array of tuples (id, distance)
3 for s ∈ S do
4     d ← Σ_{m=1}^{M} A(m, x̄_s^m)
5     PushBack(U, [s, d])
6 PartialSort(U, R) # Sort by distance
7 return Take(U, R) # Top R
```

PQ-linear-scan. : Because the database PQ-codes are stored linearly, we can simply pick up target PQ-codes and evaluate the distances to the query. We call this a PQ-linear-scan. This is essentially fast if $|\mathcal{S}|$ is small, because only a fraction of vectors are compared. The pseudocode is presented in Alg. 1.

As inputs, the system accepts a query vector $\mathbf{q} \in \mathbb{R}^D$, database PQ-codes $\bar{\mathcal{X}} = \{\bar{\mathbf{x}}_n\}_{n=1}^N$, the number of returned items $R \in \{1, \dots, N\}$, and the target identifiers $\mathcal{S} \subseteq \{1, \dots, N\}$. First, a distance table $\mathbf{A}$ is created by comparing a query to code words[3] (L1). This is an online pre-processing step, required for all PQ-based methods. To store the results, an array of tuples is prepared (L2). Each tuple consists of (1) an identifier of an item and (2) the distance between the query and the item. For each target identifier s, the asymmetric distance to the query is computed (L4). This distance is then stored in the result array with its identifier s, where the `PushBack` function is used to append an element to an array (L5). After all target items have been evaluated, the result array is sorted by the distance (L6). As we require only the top R results, we use a partial sort algorithm. Finally, the top R elements are returned, where the `Take` function simply picks up the first several elements (L7). Note that $\mathcal{W}$ and $\bar{\mathcal{C}}$ are not required for the search.

Let us analyze the computational cost. The creation of a distance table requires $O(DZ)$, and a comparison to $|\mathcal{S}|$ items requires $O(M|\mathcal{S}|)$. Partial sorting requires $O(|\mathcal{S}| \log_2 R)$ on average[4]. Their sum leads to a final average cost (Table 1). It is clear that the computation is efficient if $|\mathcal{S}|$ is small. As the cost depends on $|\mathcal{S}|$ linearly, a PQ-linear-scan becomes inefficient if $|\mathcal{S}|$ is large. Note that if the search target is the whole dataset, $|\mathcal{S}|$ is replaced by N.

Inverted-index. : The other search method is inverted-index. Because the database items are preliminarily clustered as explained in Sec. 4.1, we can simply evaluate items that are in the same/close clusters to the query. This drastically boosts the performance if the number of the target identifiers is large.

We show the pseudo-code in Alg. 2. Inverted-index takes three additional inputs: posting lists $\mathcal{W}$, coarse centers $\bar{\mathcal{C}}$, and the number

[3] We intentionally omitted the code words from the pseudocode, for simplicity.

[4] This cost comes from the heap sort-based implementation used in the `partial_sort` function in C++ STL. Another option is to pick up the k smallest items and only sort these. This leads to $O(|\mathcal{S}| + R \log_2 R)$. We used the former in this paper because we empirically determined that the former is faster in practice, especially when R is small.

Table 1: The average computational complexity for each operation. The range for each variable used in this paper: $96 \leq D \leq 960$, $Z = 256$, $8 \leq M \leq 240$, $10^6 \leq N \leq 10^9$, $1 \leq R \leq 100$, $10^2 \leq |\mathcal{S}| \leq 5 \times 10^5$, $10^3 \leq K \leq 3.2 \times 10^4$, $10^3 \leq L \leq 3.2 \times 10^4$.

Operation	Computational complexity
PQLinearScan	
- whole data	$O(DZ + MN + N \log_2 R)$
- susbet ($\mathcal{S}$)	$O(DZ + M\|\mathcal{S}\| + \|\mathcal{S}\| \log_2 R)$
InvertedIndex	
- whole data	$O\left(DZ + KM + K \log_2 \frac{KL}{N} + LM + L \log_2 R\right)$
- susbet ($\mathcal{S}$)	$O\left(DZ + KM + K \log_2 \left(\min\left(\frac{KL}{\|\mathcal{S}\|}, K\right)\right) + \frac{LN}{\|\mathcal{S}\|} \log_2 \|\mathcal{S}\| + LM + L \log_2 R\right)$

of candidates L. Note that L candidates will be selected and evaluated in the final step. This means that L is a runtime parameter that controls the trade-off between the accuracy and runtime.

To search, a distance table is first created in the same manner as for PQ-linear-scan (L1). The search steps consists of two blocks. First, the closest clusters to the query are found (L2-6). Then, the items inside the clusters are evaluated (L7-16).

To find the closest clusters, an array of tuples is created (L2). For each coarse center ($\bar{\mathbf{c}}_k$), the distance from the query is computed (L4). The results are stored in the array (L5).

Next, we run partial sort on the array to find the closest clusters to the query (L6). Here, the target number of the partial sort (the number of postings lists to be focused) is set as $\left\lceil \frac{KL}{|\mathcal{S}|} \right\rceil$, which is determined as follows. Because the target identifiers are of size $|\mathcal{S}|$, where the total number of identifiers is N, the probability of any item being a target identifier is $|\mathcal{S}|/N$ on average. Because our purpose here is to select L target items as candidates of the search, the required number of items to traverse is $L/(|\mathcal{S}|/N) = LN/|\mathcal{S}|$. To traverse $LN/|\mathcal{S}|$ items, we need to focus on $(LN/|\mathcal{S}|)/(N/K) = KL/|\mathcal{S}|$ posting lists, because the average number of items per posting list is N/K. This implies that we need to select the nearest $\left\lceil \frac{KL}{|\mathcal{S}|} \right\rceil$ posting lists. Note that if $K < \frac{KL}{|\mathcal{S}|}$, we simply replace the value by K, because this performs a full sort of the array ($O(K \log_2 K)$).

The selected posting lists are then evaluated. A score array is prepared (L7). For each closest posting list (L8), identifiers in the posting list are traversed (L9). If an identifier is not included in the target identifier $\mathcal{S}$, then this item is simply ignored (L10-11). Note that if the search is for the whole dataset ($\mathcal{S} = \{1, \ldots, N\}$), any item n is always included in $\mathcal{S}$, thus we remove L10-11.

For a selected identifier n, the identifier and the distance to the query are recorded in the same manner as for the PQ-linear-scan (L12-13). If the size of the score array ($|\mathcal{U}|$) reaches the parameter L, then the top R results are selected and returned (L14-16).

The computational cost is summarized as follows. After the code creation with $O(DZ)$, the comparison to K coarse centers requires $O(KM)$. Partial sort requires $O(K \log_2(KL/|\mathcal{S}|))$. The number of items to be traversed is $O(LN/|\mathcal{S}|)$. We can check whether or not each item is included in $\mathcal{S}$ using a binary search, requiring $O(\log_2 |\mathcal{S}|)$. This leads to $O(LN/|\mathcal{S}| \cdot \log_2 |\mathcal{S}|)$ in total. The number of items that are actually evaluated is L, and so $O(LM)$ of the cost is required. Finally, the top R are selected using the partial sort, requiring $O(L \log_2 R)$. Table 1 summarizes the computational cost. Inverted-index is fast when $|\mathcal{S}|$ is sufficiently large, but is slow if $|\mathcal{S}|$ is small. This is highlighted in the term $LN/|\mathcal{S}| \log_2 |\mathcal{S}|$, where this term becomes dominant if $|\mathcal{S}|$ is small.

Algorithm 2: `InvertedIndex`

Input: $\mathbf{q} \in \mathbb{R}^D$, # Query
$\bar{\mathcal{X}} = \{\bar{\mathbf{x}}_n\}_{n=1}^N$, # Database PQ-codes
$\mathcal{W} = \{\mathcal{W}_k\}_{k=1}^K$, # Posting lists
$\bar{\mathcal{C}} = \{\bar{\mathbf{c}}_k\}_{k=1}^K$, # Coarse centers
$R \in \{1, \ldots, N\}$, # # of returned items
$\mathcal{S} \subseteq \{1, \ldots, N\}$, # Target subset identifiers
$L \in \{1, \ldots, N\}$ # # of candidates
Output: $\mathcal{U} = \{\mathbf{u}_r\}_{r=1}^R$ s.t. $\mathbf{u}_r = [n_r, d_r] \in \{1, \ldots, N\} \times \mathbb{R}$
n_r: r-th identifier. d_r: r-th distance.

```
1  A ← CompareCodewords(q) # Distance table
2  T ← ∅ # Array of tuples (id, distance)
3  for k ∈ {1, ..., K} do
4      d_0 ← Σ_{m=1}^{M} A(m, c̄_k^m)
5      PushBack(T, [k, d_0])
6  PartialSort(T, ⌈KL/|S|⌉) # Sort by distance
7  U ← ∅ # Array of tuples (id, ditance)
8  for [k, d_0] ∈ T do
9      for n ∈ W_k do
10         if n ∉ S then
11             continue
12         d ← Σ_{m=1}^{M} A(m, x̄_n^m)
13         PushBack(U, [n, d])
14         if |U| = L then
15             PartialSort(U, R) # Sort by distance
16             return Take(U, R) # Top R
```

Note that although there appear to be several input parameters for inverted-index, all of them except L are usually decided deterministically. L is the only parameter the user needs to decide. Our initial setting is the average length of a posting list, $L = N/K$. This means that the system traverses one posting list on average. This is a fast setting, and users can change this if they require more accuracy, as $L = 2N/K, 3N/K, \ldots$.

Selection. : The final query algorithm is described in Alg. 3. Given inputs, the system automatically determines the query method as either PQ-linear-scan or inverted-index. This decision is based on the threshold value θ for the number of target identifiers (L1). Owing to this flexible switching, we can always achieve a fast search with a single Rii data structure ($\bar{\mathcal{X}}$, $\mathcal{W}$, and $\bar{\mathcal{C}}$), regardless of the sizes of the target identifiers ($|\mathcal{S}|$). Fig. 2 highlights the relations among the three query algorithms.

Note that it is not trivial to set the threshold θ deterministically, because it depends on several parameters, such as M and L. To find the best threshold, we simply run the search with several parameter combinations when the data structure is constructed. Based on the result, we fit a 1D line in the parameter space, and finally obtain the best threshold. See the supplementary material for more details. This works almost perfectly, as shown in Fig. 2. This thresholding does not require any additional runtime cost for the search phase.

Algorithm 3: Query

Input: $\mathbf{q}, \bar{\mathcal{X}}, \mathcal{W}, \bar{\mathcal{C}}, R, \mathcal{S}, L$
See the definitions in Alg. 2
Output: $\mathcal{U} = \{\mathbf{u}_r\}_{r=1}^{R}$ s.t. $\mathbf{u}_r = [n_r, d_r] \in \{1, \dots, N\} \times \mathbb{R}$
1 **if** $|\mathcal{S}| < \theta$ **then**
2 **return** PQLinearScan($\mathbf{q}, \bar{\mathcal{X}}, R, \mathcal{S}$) # Alg. 1
3 **else**
4 **return** InvertedIndex($\mathbf{q}, \bar{\mathcal{X}}, \mathcal{W}, \bar{\mathcal{C}}, R, \mathcal{S}, L$) # Alg. 2

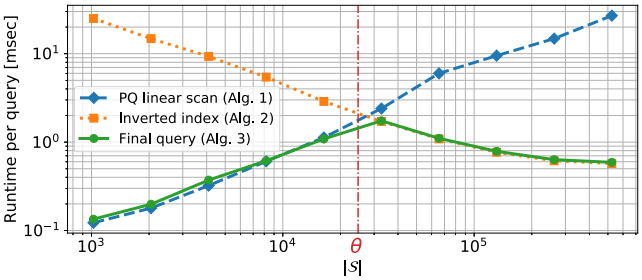

Figure 2: Comparison of PQ-linear-scan, inverted-index, and the final query algorithm. Runtime per query for the SIFT1M dataset with various sizes of target identifiers is plotted. Note that $L = K = 1000, R = 1, \theta = 24743$.

4.3 Reconfiguration

Here, we introduce a reconfigure function that enables us to search efficiently even if a large number of vectors are newly added. As discussed in Sec. 1, typical ANN systems are first optimized to achieve fast searching for N items. If new items are added later, such systems might become slow. For example, IVFADC requires an initial decision on the number of space partitions K. The selection of K is sensitive and critical to the performance. A standard convention[5] is to set $K = \sqrt{N}$. On the other hand, K cannot be changed later. The system could become slower if N changes significantly. In other words, we must decide K even if the final database size N is not known, which sometimes frustrates users.

Unlike these existing methods, Rii provides a reconfigure function. If the search becomes slow because of newly added items, coarse centers and assignments are updated by simply running clustering again. The system is automatically optimized to achieve the fastest search for the current number of database items.

Data addition. Let us first explain how to add a new item. Given a new PQ-code $\bar{\mathbf{y}}$, the database PQ-codes $\bar{\mathcal{X}} = \{\bar{\mathbf{x}}_n\}_{n=1}^{N}$ are updated using PushBack($\bar{\mathcal{X}}, \bar{\mathbf{y}}$). A corresponding posting list is also updated by PushBack($\mathcal{W}_{a(N+1)}, N+1$). Then, searching can be performed without any modifications, but it may be slower if many items are added. This is because the length of each posting list ($|\mathcal{W}_k|$) can become too long, making the traversal inefficient.

Reconfigure. If the search becomes slow, a reconfigure function can be called (Alg. 4). This function takes the database PQ-codes $\bar{\mathcal{X}}$ and a new number of coarse space partitions K' as inputs. Again, K' is typically set as $\sqrt{N}$ for the new N. The outputs are updated posting lists and coarse centers. First, the updated coarse centers

[5]https://github.com/facebookresearch/faiss/wiki/Index-IO,-index-factory,-cloning-and-hyper-parameter-tuning

Algorithm 4: Reconfigure

Input: $\bar{\mathcal{X}} = \{\bar{\mathbf{x}}_n\}_{n=1}^{N}$, # Database PQ-codes
$K' \in \{1, \dots, N\}$ # # of coarse centers
Output: $\mathcal{W} = \{\mathcal{W}_k\}_{k=1}^{K'}$, # Updated posting list
$\bar{\mathcal{C}} = \{\bar{\mathbf{c}}_k\}_{k=1}^{K'}$ # Updated coarse centers
1 $\bar{\mathcal{C}} \leftarrow$ PQkmeans($\bar{\mathcal{X}}, K'$) # Clustering on PQ-codes [35]
2 $\mathcal{W} \leftarrow \emptyset$
3 **for** $k \in \{1, \dots, K'\}$ **do**
4 $\mathcal{W}_k \leftarrow \{n \in \{1, \dots, N\} | a(n) = k\}$
5 **return** $\mathcal{W}, \bar{\mathcal{C}}$

are computed by running clustering over the PQ-codes using PQk-means [35] (L1). PQk-means efficiently puts the input PQ-codes into several clusters, without decoding the codes for the original D-dimensional vectors. Note that clustering can be run for a subset of $\bar{\mathcal{X}}$ to make this fast. We set the upper limit of the codes to be clustered as $\min(N, 100K')$. After new coarse centers are obtained, the posting lists are created by simply finding the nearest center for each PQ-code (L2-4).

The advantage of the reconfigure function is that it can be called whenever the user wishes. The results are deterministic for K', because this just runs the clustering over the codes. We will show in Sec. 5.4 that this reconfigure function is especially useful when the database size drastically changes. Another way of looking at this is that we do not need to know the final number of database items when the index structure is built. This is a clear advantage over IVFADC-based methods. In a practical scenario, it will often occur that the number of database items cannot be decided when the system is created. Even in such cases, IVFADC must decide the parameters. This would lead to a suboptimal performance.

4.4 Connection to IVFADC

The data structure proposed above is similar to the original IVFADC [26], but has the following fundamental differences.

- In Rii, each vector is encoded directly, whereas IVFADC encodes a residual between an input vector and a coarse center. This makes the accuracy of Rii slightly inferior to that of IVFADC (see Sec. 5.5), but enables us to store PQ-codes linearly.
- In Rii, PQ-codes are stored linearly, and their identifiers are stored in posting lists. In IVFADC, both PQ-codes and identifiers are stored in posting lists. This simple modification enables us to run the PQ-linear scan without any additional operations.
- In IVFADC, coarse centers are a set of D-dimensional vectors, whereas coarse centers in Rii are PQ-codes. The advantage of this is that the reconfigure steps become considerably fast with PQk-means. The limitation is that this might decrease the accuracy, but the experimental results show that this degradation is not serious (Sec. 5.5).

4.5 Advanced Encoding

There exist advanced encoding methods for PQ, such as optimized product quantization (OPQ) [18, 40], additive quantization (AQ) [6, 34], and composite quantization (CQ) [48, 49]. Although state-of-the-art accuracy has been achieved by AQ or CQ, it is widely known

that they are more complex and time consuming. Therefore, we did not incorporate AQ and CQ in our system.

On the other hand, OPQ provides a reasonable trade-off (slightly slow but with a high accuracy). In OPQ, a rotation matrix is preliminarily trained to minimize the error. In the search phase, an input vector is first rotated with the matrix. The remaining process is exactly the same as PQ. We will show the results of OPQ in Sec. 5.5.

5 EVALUATIONS

All experiments were performed on a server with a 3.6 GHz Intel Xeon CPU (six cores, 12 threads) and 128 GB of RAM. For a fair comparison, we employed a single-thread implementation for the search. Rii is implemented by C++ with a Python interface, All source codes are publicly available[6]

5.1 Datasets

The various methods were evaluated using the following datasets:

- SIFT1M [27] consists of 128D SIFT feature vectors extracted from several images. It provides 1,000,000 base, 10,000 query, and 100,000 training vectors.
- GIST1M [27] consists of 960D GIST feature vectors extracted from several images. It provides 1,000,000 base, 1,000 query, and 500,000 training vectors.
- Deep1B [9] consists of 96D deep features extracted from the last FC layer of GoogLeNet [43] for one billion images. It provides 1,000,000,000 base, 10,000 query, and 1,000,000 (we used the top 1M from the whole training branch) training vectors.

The code words of Rii and Faiss were preliminarily trained using the training data. The search is conducted over the base vectors.

5.2 Methods

We compare our Rii method with the following existing methods:

- Annoy [11]: A random projection forest-based system. Because Annoy is easy to use (fewer parameters, intuitive interface, no training steps, and easy IO with a direct mmap design), it is the baseline for million-scale data.
- FALCONN [41]: Highly optimized LSH [1]. FALCONN is regarded as a representative state-of-the-art LSH-based method.
- NMSLIB [39]: Highly optimized ANN library with the support of non-metric spaces [14]. This library includes several algorithms, and we used Hierarchical Navigable Small World (HNSW) [32, 33] in this study. NMSLIB with HNSW is the current state-of-the-art for million-scale data [4, 12].
- Faiss [25]: A collection of highly-optimized PQ-based methods. This library includes IVFADC [26], OPQ [18], inverted multi-index [7], and polysemous codes [16]. Some of these are implemented using the GPU as well [28]. In particular, we compared Rii with the basic IVFADC, which is one of the fastest options. Note that only Faiss and Rii can handle billion-scale data, because PQ-based methods are memory efficient.

5.3 Subset Search

We first present the results for searching over a subset of the whole database. This is the main function that the proposed Rii method provides. The conclusion is that Rii always remains fast, whereas existing methods become considerably slower, especially if the size of the target subset is small. We first explain the task, and then introduce a post-checking module through which existing methods can conduct a subset search. Finally, we present the results.

Task. The task is defined as follows. We randomly select integers from $\{1, \dots, N\}$, sort them, and construct the target indices $\mathcal{S} \subseteq \{1, \dots, N\}$. For each query, we run the search and find the top-R results. All the results must be members of $\mathcal{S}$. The runtime per query was reported with several combinations of $\mathcal{S}$ and R. The evaluation was conducted using the SIFT1M dataset ($N = 10^6$), with $R \in \{1, 10, 100\}$.

Post-checking module. Because none of the existing methods provide a subset search functionality, we implemented a straightforward post-checking module in order to enable the existing methods to perform a subset search. Alg. 5 shows the pseudocode. This module takes a query function Q, a query vector $\mathbf{q}$, target identifiers $\mathcal{S}$, and the number of returned items R as inputs. The query function Q returns the identifiers of the R closest items, given $\mathbf{q}$ and R. This Q is an existing method such as Annoy. First, the output identifier set is prepared (L1). The number of returned items for each iteration, r, is first initialized (L2). Then, the search begins with an infinite loop. The top-r items are searched using Q, and the results are stored in the temporal buffer $\mathcal{T}$ (L4). For each identifier n in $\mathcal{T}$, if n has already been checked, the loop continues (L6-7). This is actually achieved by starting a for loop with some offsets over $\mathcal{T}$, so that the first already-checked elements up to a certain number are not traversed. If n is included in $\mathcal{S}$, we store it in the output set $\mathcal{U}$ (L8-9). The algorithm finishes if the enough (R) items are found (L10-11). If an insufficient number of items are found, then r is updated to a larger number by simply multiplying a constant value (L12). The search continues with the updated r until R items are found.

With this module, searching over a target subset is made available for the existing methods. Note that Q cannot always return r items when r is large. This depends on the design of the query function, and some methods have a limit on r in order not to make the search too slow. We found that FALCONN and NMSLIB do not return r items if r is large. Therefore, we compared Rii with Annoy using the post-checking module (Annoy + PC).

Results. Fig. 3 illustrates the results. We point out the following:

- Rii was fast under all conditions (less than 2 ms/query). We can conclude that Rii was stable and effective for the subset-search.
- As with IVFADC, Rii is robust against R.
- Annoy + PC became drastically slow for small $|\mathcal{S}|$, which is further highlighted when R is large. This is an obvious result, because the while loop (L3 in Alg. 5) must be repeated several times for large r. Here, r can be even N. ANN systems are usually not designed to handle such r values.

5.4 Robustness Against Data Addition

We describe the experiments for our other main function, reconfigure. The conclusion is that Rii becomes fast by using reconfigure, even after many new vectors are added. First, the task is explained, then the results are presented. Here, we used the Deep1B dataset to demonstrate the robustness against billion-scale data.

[6] https://github.com/matsui528/rii

Algorithm 5: Post-checking module for existing methods.

Input: Q, # Query function
$\mathbf{q} \in \mathbb{R}^D$, # Query vector
$\mathcal{S} \subseteq \{1, \dots, N\}$, # Target subset identifiers
$R \in \{1, \dots, N\}$, # # of returned items
Output: $\mathcal{U} \subseteq \mathcal{S}$ # $\mathcal{U}$ is sorted

$\mathcal{U} \leftarrow \emptyset$ # An array of integers
$r \leftarrow R$
while *1* **do**
  $\mathcal{T} \leftarrow \mathrm{Q}(\mathbf{q}, r)$ # Return top-r results
  for $n \in \mathcal{T}$ **do**
    if *n has been already checked* **then**
      continue
    if $n \in \mathcal{S}$ **then**
      PushBack($\mathcal{U}, n$)
    if $|\mathcal{U}| = R$ **then**
      return $\mathcal{U}$
  $r \leftarrow r \times 5$ # User defined constant value

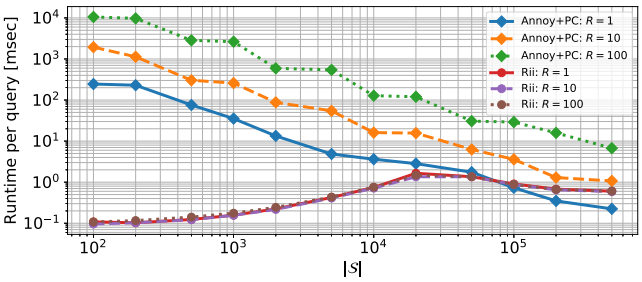

Figure 3: Subset search using the SIFT1M dataset over 10 queries. Note that $K = L = 1000, M = 64$.

Task. The index is first constructed using $N = 10^6$ vectors with $K = \sqrt{N} = 10^3$, and then the runtime is evaluated. Next, new items are added to the index, so that the final N becomes 10^7. Then, the runtime is evaluated in two ways: (1) a search is performed with $K = 10^3$, and (2) the data structure is updated using the reconfigure function with $K = \sqrt{10^7}$, and then the search is conducted. We run this experiment with the final N as 10^7, 10^8, and 10^9.

Results. Fig. 4 illustrates the result. It is clear that the search becomes dramatically faster after the reconfigure function is called. For example, if the user keeps the same data structure after $99M$ new items are added, the search takes an average of 3.9 ms. This can be made 7.8× faster after applying the reconfigure function.

Most importantly, because the data structure can be always adjusted for the new N, the user need not face the burden of selecting K when the system is constructed. This is a clear advantage over the other existing methods. Note that the runtime for adding 9×10^6 vectors was 109 s, and that of the reconfigure function with $K = \sqrt{10^7}$ was 111 s. These times can be considered moderate.

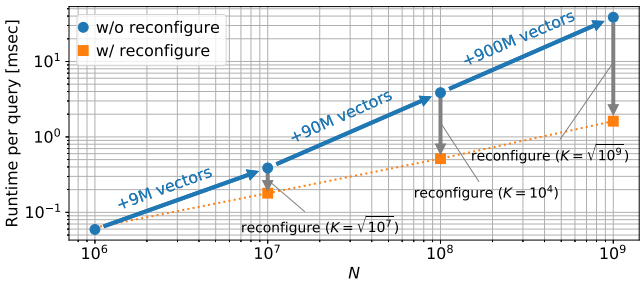

Figure 4: The runtime performance with and without the reconfigure function over the Deep1B dataset, where $R = 1$, $M = 8$, and $L = N/K$.

5.5 Comparison with Existing Methods

Finally, we compare Rii (and its variant Rii-OPQ) with Annoy, FALCONN, NMSLIB (HNSW), and Faiss (IVFADC), using SIFT1M and GIST1M. The conclusion is that our Rii method achieved a comparable performance to the state-of-the-art system Faiss. Note that the searches were conducted over the whole datasets.

The accuracy was measured using Recall@1, which measures the fraction of queries for which the ground truth nearest neighbor is returned within the top-1 result. The average Recall@1 over the query set is reported. We evaluated the methods with several parameter combinations, and report the results with a fixed Recall@1 (0.65 for SIFT1M and 0.5 for GIST1M) for a fair comparison. Because the ranges of some parameters are discrete, we cannot achieve an exact target Recall@1. Thus, the target Recall@1 was selected as best as possible as a value that all methods can achieve.

The disk consumption of the index data structure is also reported. This was measured by storing the data structure on the disk and checking its size in bytes. Note that the runtime (peak-time) memory consumption is the more important measure, but measuring the peak-time memory usage is not always stable, and can vary depending on the computer. Thus, we report the disk space instead, which is reproducible and strongly related to the memory consumption. The runtime of building the data structure is also reported.

Table 2 presents the results. We summarize our findings:

- Rii was comparable with the state-of-the-art system Faiss. In particular, although our method is basically an approximation of IVFADC, the decrease in the accuracy is not significant.
- Rii was the most memory efficient among the methods. The measured value is almost same as the theoretically predicted value (68 MB against 69 MB and 244 MB against 249 MB).
- If we compare Rii and Rii-OPQ, Rii-OPQ was slightly slower but a little more accurate with the same parameter settings.
- Annoy achieved the second fastest result. Because Annoy supports the direct memory map system, the construction required some time and consumed a relatively large disk space.
- FALCONN achieved a comparable (or slightly slower) performance to Faiss/Rii. We note that the building cost of FALCONN is considerably smaller than for other methods. As FALCONN does not provide IO functions, we did not report the disk space.
- As reported in the benchmark [4, 12], NMSLIB achieved the fastest performance. On the other hand, the building time and memory consumption are inferior relative to Faiss/Rii.
- The results for SIFT1M and GIST1M follow similar tendencies.

Table 2: Comparison to existing methods using SIFT1M/GIST1M. Note that $R = 1$ for all methods. Unless explicitly denoted, we adopt the default parameters for each method. The bold fonts indicate the best scores among the methods.

Dataset	Method	Parameters	Recall@1 (fixed)	Runtime/query	Disk space	Build time
SIFT1M	Annoy [11]	$n_{\text{trees}} = 2000$, $k_{\text{search}} = 400$	0.67	0.18 ms	1703 MB	899 sec
	FALCONN [1, 41]	$n_{\text{probes}} = 16$	0.63	0.87 ms	-	**1.8 sec**
	NMSLIB (HNSW) [14, 33, 39]	efS = 4	0.67	**0.043 ms**	669 MB	436 sec
	Faiss (IVFADC) [25, 26]	$K = 10^3, M = 64, n_{\text{probe}} = 4$	0.67	0.61 ms	73 MB	30 sec
	Rii (proposed)	$K = 10^3, M = 64, L = 5000$	0.64	0.73 ms	**69 MB**	82 sec
	Rii-OPQ (proposed)	$K = 10^3, M = 64, L = 5000$	0.65	0.82 ms	**69 MB**	85 sec
GIST1M	Annoy [11]	$n_{\text{trees}} = 2000$, $k_{\text{search}} = 2000$	0.49	1.2 ms	5023 MB	2088 sec
	FALCONN [1, 41]	$n_{\text{probes}} = 512$	0.53	8.6 ms	-	**7.2 sec**
	NMSLIB (HNSW) [14, 33, 39]	efS = 8	0.49	**0.19 ms**	3997 MB	1576 sec
	Faiss (IVFADC) [25, 26]	$K = 10^3, M = 240, n_{\text{probe}} = 8$	0.52	3.8 ms	253 MB	51 sec
	Rii (proposed)	$K = 10^3, M = 240, L = 8000$	0.45	3.2 ms	**246 MB**	353 sec
	Rii-OPQ (proposed)	$K = 10^3, M = 240, L = 8000$	0.50	3.8 ms	249 MB	388 sec

Table 3: Metadata of MET dataset. Each item has several attributes, such as title and data.

ID	title	date	country	···
0	Bust of Abraham Lincoln	1876	United States	
1	Acorn Clock	1847	United States	
	⋮			

6 APPLICATION

We present an application to highlight the subset search function of Rii. For this demonstration, we leverage the data of The Metropolitan Museum of Art (MET) Open Access[7]. This dataset contains more than 420,000 items from MET, with both the image and extensive metadata for each item (Table 3). From this data, we select 201,998 items that are provided with the Creative Common license. For each image, we extracted a 1,920-dimensional activation of last average pooling layer of the DenseNet-201 [22] architecture trained with ImageNet. The features are stored in Rii with $M = 192$. Several meta-information is stored in a table using Pandas[8], which is a popular on-memory data management system for Python.

Fig. 5 demonstrates the system, including Python codes and the search results. The metadata and DenseNet vectors are first read. Then, the search is conducted based on the metadata. Here, the items that were created before A.D. 500 in Egypt are specified. Next, the target identifiers $\mathcal{S}$ are prepared. This is simply a set of IDs of the selected items. The image-based search is then conducted over them. The query here is Chinese tapestry. We can find similar items to the Chinese tapestry from the museum items in ancient Egypt.

As this demonstration reveals, the search using the target subset is a general problem setting. Rii can solve this type of problem easily. As Sec. 5.3 shows, existing methods using the late checking module do not perform well when $|\mathcal{S}|$ is small. For example, in this case the result of the metadata search can have any number of items. Rii can handle a subset search for any size of $\mathcal{S}$.

[7]https://github.com/metmuseum/openaccess
[8]https://pandas.pydata.org/

```
import pandas as pd
import rii

# Read data
df = pd.read_csv('metadata.csv')
engine = pkl.load(open('rii_densenet.pkl', 'rb'))

# Metadata search (13.5 ms)
S = df[(df['data']<500) & (df['country']=='Egypt')]['ID']
S = np.sort(np.array(S))  # Target identifiers

# ANN for subset (2 ms)
q = # Read query feature
result = engine.query(q=q, target_ids=S, topk=3)
```

Query The nearest The 2nd nearest The 3rd nearest

Figure 5: Demonstration of the subset search. The target items are first selected using metadata information. Then, an image-based search is conducted over the target items.

7 CONCLUSIONS

We developed an approximate nearest neighbor search method, called Rii. Rii provides the two functions of searching over a subset and a reconfigure function for newly added vectors. Extensive comparisons showed that Rii achieved a comparable performance to state-of-the art systems, such as Faiss.

Note that the latest systems incorporate HNSW for the coarse assignment of IVFADC [10, 17]. Our Rii architecture can be combined to them, but that will be remained as a future work.

Acknowledgments: This work was supported by JST ACT-I Grant Number JPMJPR16UO, Japan.

REFERENCES

[1] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. 2015. Practical and Optimal LSH for Angular Distance. In *Proc. NIPS*.

[2] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2015. Cache Locality is Not Enough: High-performance Nearest Neighbor Search with Product Quantization Fast Scan. In *Proc. VLDB*.

[3] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2017. Accelerated Nearest Neighbor Search with Quick ADC. In *Proc. ICMR*.

[4] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2017. ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms. In *Proc. SISAP*.

[5] Artem Babenko and Victor Lemitsky. 2017. AnnArbor: Approximate Nearest Neighbors Using Arborescence Coding. In *Proc. IEEE ICCV*.

[6] Artem Babenko and Victor Lempitsky. 2014. Additive Quantization for Extreme Vector Compression. In *Proc. IEEE CVPR*.

[7] Artem Babenko and Victor Lempitsky. 2015. The Inverted Multi-Index. *IEEE TPAMI* 37, 6 (2015), 1247–1260.

[8] Artem Babenko and Victor Lempitsky. 2015. Tree Quantization for Large-Scale Similarity Search and Classification. In *Proc. IEEE CVPR*.

[9] Artem Babenko and Victor Lempitsky. 2016. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In *Proc. IEEE CVPR*.

[10] Dmitry Baranchuk, Artem Babenko, and Yury Malkov. 2018. Revisiting the Inverted Indices for Billion-Scale Approximate Nearest Neighbors. In *Proc. ECCV*.

[11] Erik Bernhardsson. 2018. Annoy. https://github.com/spotify/annoy.

[12] Erik Bernhardsson, Martin Aumüller, and Alexander Faithfull. 2018. ann-benchmarks. https://github.com/erikbern/ann-benchmarks.

[13] Davis W. Blalock and John V. Guttag. 2017. Bolt: Accelerated Data Mining with Fast Vector Compression. In *Proc. ACM KDD*.

[14] Leonid Boytsov and Bilegsaikhan Naidan. 2013. Engineering Efficient and Effective Non-metric Space Library. In *Proc. SISAP*.

[15] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-Sensitive Hashing Scheme Based on p-Stable Distributions. In *Proc. SCG*.

[16] Matthijs Douze, Hervé Jégou, and Florent Perronnin. 2016. Polysemous Codes. In *Proc. ECCV*.

[17] Matthijs Douze, Alexandre Sablayrolles, and Hervé Jégou. 2018. Link and code: Fast indexing with graphs and compact regression codes. In *Proc. IEEE CVPR*.

[18] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2014. Optimized Product Quantization. *IEEE TPAMI* 36, 4 (2014), 744–755.

[19] Gylfi Þór Gudmundsson, Björn Þór Jónsson, Laurent Amsaleg, and Michael J. Franklin. 2018. Prototyping a Web-Scale Multimedia Retrieval Service Using Spark. *ACM TOMM* 14, 3s (2018), 65:1–65:24.

[20] Jae-Pil Heo, Zhe Lin, Xiaohui Shen, Jonathan Brandt, and Sung-Eui Yoon. 2016. Shortlist Selection With Residual-Aware Distance Estimator for K-Nearest Neighbor Search. In *Proc. IEEE CVPR*.

[21] Jae-Pil Heo, Zhe Lin, and Sung-Eui Yoon. 2014. Distance Encoded Product Quantization. In *Proc. IEEE CVPR*.

[22] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2017. Densely Connected Convolutional Networks. In *Proc. IEEE CVPR*.

[23] Masakazu Iwamura, Tomokazu Sato, and Koichi Kise. 2013. What Is the Most Efficient Way to Select Nearest Neighbor Candidates for Fast Approximate Nearest Neighbor Search?. In *Proc. IEEE ICCV*.

[24] Himalaya Jain, , Patrick Pérez, Rémi Gribonval, Joaquin Zepeda, and Hervé Jégou. 2016. Approximate Search with Quantized Sparse Representations. In *Proc. ECCV*.

[25] Hervé Jégou, Matthijs Douze, and Jeff Johnson. 2018. Faiss. https://github.com/facebookresearch/faiss.

[26] Hervé Jégou, Matthijis Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE TPAMI* 33, 1 (2011), 117–128.

[27] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. 2011. Searching in One Billion Vectors: Re-rank with Source Coding. In *Proc. IEEE ICASSP*.

[28] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2017. Billion-scale Similarity Search with GPUs. *CoRR* abs/1702.08734 (2017).

[29] Yannis Kalantidis and Yannis Avrithis. 2014. Locally Optimized Product Quantization for Approximate Nearest Neighbor Search. In *Proc. IEEE CVPR*.

[30] Yingfan Liu, Hong Cheng, and Jiangtao Cui. 2017. PQBF: I/O-Efficient Approximate Nearest Neighbor Search by Product Quantization. In *Proc. CIKM*.

[31] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search. In *Proc. VLDB*.

[32] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate Nearest Neighbor Algorithm Based on Navigable Small World Graphs. *Inf. Syst.* 45 (2014), 61–68.

[33] Yury A. Malkov and Dmitry A. Yashunin. 2016. Efficient and Robust Approximate Nearest Neighbor Search using Hierarchical Navigable Small World Graphs. *CoRR* abs/1603.09320 (2016).

[34] Julieta Martinez, Joris Clement, Holger H. Hoos, and James J. Little. 2016. Revisiting Additive Quantization. In *Proc. ECCV*.

[35] Yusuke Matsui, Keisuke Ogaki, Toshihiko Yamasaki, and Kiyoharu Aizawa. 2017. PQk-means: Billion-scale Clustering for Product-quantized Codes. In *Proc. MM*.

[36] Yusuke Matsui, Yusuke Uchida, Hervé Jégou, and Shin'ichi Satoh. 2018. A Survey of Product Quantization. *ITE Transactions on Media Technology and Applications* 6, 1 (2018), 2–10.

[37] Yusuke Matsui, Toshihiko Yamasaki, and Kiyoharu Aizawa. 2018. PQTable: Non-exhaustive Fast Search for Product-quantized Codes using Hash Tables. *IEEE TMM* 20, 7 (2018), 1809–1822.

[38] Marius Muja and David G. Lowe. 2014. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *IEEE TPAMI* 36, 11 (2014), 2227–2240.

[39] Bilegsaikhan Naidan, Leonid Boytsov, Yury Malkov, David Novak, and Ben Frederickson. 2018. Non-Metric Space Library (NMSLIB). https://github.com/searchivarius/nmslib.

[40] Mohammad Norouzi and David J. Fleet. 2013. Cartesian k-means. In *Proc. IEEE CVPR*.

[41] Ilya Razenshteyn and Ludwig Schmidt. 2018. FALCONN - FAst Lookups of Cosine and Other Nearest Neighbors. https://github.com/FALCONN-LIB/FALCONN.

[42] Eleftherios Spyromitros-Xioufis, Symeon Papadopoulos, Ioannis (Yiannis) Kompatsiaris, Grigorios Tsoumakas, and Ioannis Vlahavas. 2014. A Comprehensive Study Over VLAD and Product Quantization in Large-Scale Image Retrieval. *IEEE TMM* 16, 6 (2014), 1713–1728.

[43] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper With Convolutions. In *Proc. IEEE CVPR*.

[44] Jianfeng Wang, Jingdong Wang, Jingkuan Song, Xin-Shun Xu, Heng Tao Shen, and Shipeng Li. 2015. Optimized Cartesian K-Means. *IEEE TKDE* 27, 1 (2015), 180–192.

[45] Patrick Wieschollek, Oliver Wang, Alexander Sorkine-Hornung, and Hendrik P. A. Lensch. 2016. Efficient Large-Scale Approximate Nearest Neighbor Search on the GPU. In *Proc. IEEE CVPR*.

[46] Yan Xia, Kaiming He, Fang Wen, and Jian Sun. 2013. Joint Inverted Indexing. In *Proc. IEEE ICCV*.

[47] Jialiang Zhang, Soroosh Khoram, and Jing Li. 2018. Efficient Large-Scale Approximate Nearest Neighbor Search on OpenCL FPGA. In *Proc. IEEE CVPR*.

[48] Ting Zhang, Chao Du, and Jingdong Wang. 2014. Composite Quantization for Approximate Nearest Neighbor Search. In *Proc. ICML*.

[49] Ting Zhang, Guo-Jun Qi, Jinhui Tang, and Jingdong Wang. 2015. Sparse Composite Quantization. In *Proc. IEEE CVPR*.