RAIRS: Optimizing Redundant Assignment and List Layout for IVF-Based ANN Search

ZEHAI YANG and SHIMIN CHEN*, SKLP, ACS, Institute of Computing Technology, CAS, China and University of Chinese Academy of Sciences, China

IVF is one of the most widely used ANNS (Approximate Nearest Neighbors Search) methods in vector databases. The idea of redundant assignment is to assign a data vector to more than one IVF lists for reducing the chance of missing true neighbors in IVF search. However, the naïve strategy, which selects the second IVF list based on the distance between a data vector and the list centroids, performs poorly. Previous work focuses only on the inner product distance, while there is no optimized list selection study for the most popular Euclidean space. Moreover, the IVF search may access the same vector in more than one lists, resulting in redundant distance computation and decreasing query throughput.

In this paper, we present RAIRS to address the above two challenges. For the challenge of the list selection, we propose an optimized AIR metric for the Euclidean space. AIR takes not only distances but also directions into consideration in order to support queries that are closer to the data vector but father away from the first chosen list's centroid. For the challenge of redundant distance computation, we propose SEIL, an optimized list layout that exploits shared cells to reduce repeated distance computations for IVF search. Our experimental results using representative real-world data sets show that RAIRS out-performs existing redundant assignment solutions and achieves up to 1.33x improvement over the best-performing IVF method, IVF-PQ Fast Scan with refinement.

CCS Concepts: • **Information systems** → **Top-k retrieval in databases**; **Data access methods**.

Additional Key Words and Phrases: Vector Database; IVF Index; Approximate Nearest Neighbor Search.

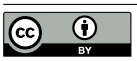
ACM Reference Format:
Zehai Yang and Shimin Chen. 2026. RAIRS: Optimizing Redundant Assignment and List Layout for IVF-Based ANN Search. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 73 (February 2026), 27 pages. https://doi.org/10.1145/3786687

1 Introduction

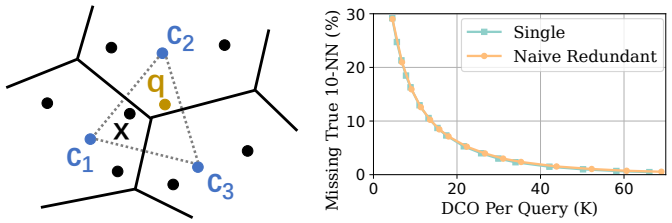
Vector search is widely used in real-world applications, including recommendations [46], data mining [14], information retrieval [37], and recently large language model (LLM) studies [12, 61]. Approximate Nearest Neighbors Search (ANNS) is the key operation in vector databases. The inverted file index (IVF) is one of the most widely adopted ANNS methods. As depicted in Figure 6a, during construction, IVF computes *nlist* clusters (a.k.a. lists) and assigns each data vector to the list whose centroid is the closet to the vector. For example, x is assigned to $list_1$ as x lies closer to c_1 than any other centroids. Given a query q, IVF searches the list centroids and chooses the top-*nprobe* nearest lists to q, then traverses the chosen lists to compute distance for all vectors in the lists. In this example, IVF chooses the top-2 lists (i.e., $list_2$ and $list_3$). Unfortunately, it fails to retrieve x, q's true nearest neighbor. While increasing *nprobe* (e.g., from 2 to 3) may help, IVF

*Shimin Chen is the corresponding author.

Authors' Contact Information: Zehai Yang, yangzehai20z@ict.ac.cn; Shimin Chen, chensm@ict.ac.cn, SKLP, ACS, Institute of Computing Technology, CAS, Beijing, China and University of Chinese Academy of Sciences, Beijing, China.



ACM 2836-6573/2026/2-ART73
https://doi.org/10.1145/3786687



(a) Problem of single assignment (b) NaïveRA works poorly

Fig. 1. Redundant assignment for IVF index.

has to traverse more lists, leading to lower query throughput. In this paper, we investigate an alternative solution, redundant assignment. The idea is to assign a data vector to more than one IVF lists. Suppose x is assigned to both $list_1$ and $list_2$. Then, the traversal of $list_2$ can successfully retrieve x, thereby reducing the chance of missing true neighbors.

Challenges of Redundant Assignment. There are two main challenges for realizing the redundant assignment idea:

- *List selection strategy*: The naïve strategy (NaïveRA) is to select the second list for a vector based purely on its distance to the list centroids. However, experimental results show that NaïveRA can hardly improve the ANNS performance. Figure 6b compares NaïveRA with single assignment on the SIFT1M data set for top-10 nearest-neighbor search. The X-axis reports the average distance computing operations (DCO) per query, while the Y-axis shows the percentage of true neighbors missed. We see that the two curves almost overlap, indicating that NaïveRA fails to out-perform the baseline IVF single assignment.
- *Redundant distance computation*: With redundant assignment, IVF search may encounter the same vector in more than one chosen lists. For example, suppose x is assigned to $list_1$ and $list_2$. If both lists are chosen in a query, IVF will access x twice, leading to redundant distance computation for x. One fix seems to deduplicate the vector IDs from all chosen lists before distance computation. However, in advanced IVF methods, such as IVF-PQ fast scan [3], lists store the vector IDs in packed blocks to facilitate SIMD acceleration. It is very costly to unpack the blocks to obtain individual IDs for deduplication purposes.

Our Solution: RAIRS. To address these challenges, we propose RAIRS, consisting of the following two optimization techniques:

- *RAIR (Redundant list selection with Amplified Inverse Residual).* We propose an optimized AIR metric for secondary list selection in the Euclidean space. After assigning a vector to the first list, whose centroid is the closest, RAIR selects the second list to minimize the AIR metric, i.e., $(||r'||^2+\lambda r^T r')$, where r, r' are the clustering residuals of the first and the second lists, respectively, and λ is a constant parameter. This metric considers not only the distance (i.e., the first term $||r'||^2$), but also the direction (i.e., the second term $r^T r'$). AIR prefers a negative second term; it selects the second list whose centroid is at an inverse direction of the data vector compared to the first assigned list's centroid, thereby covering queries that are closer to the vector but father away from the first assigned list's centroid. We formally prove the effectiveness of AIR. In addition, we investigate multiple list assignment by extending RAIR to select three or more lists.
- *SEIL (Shared-Cell Enhanced IVF Lists).* To alleviate redundant distance computation, we propose an optimized list layout, SEIL. We use $cell_{i,j}$ to denote all vectors that are assigned to both $list_i$ and $list_j$. Based on real-world data analysis, we observe that there are cells that contain a large number of vectors. In advanced IVF methods, such as IVF-PQ fast scan, every 32 vector items are packed into a block for SIMD acceleration. Thus, we call cells larger than a block as large cells.

We observe that a large fraction of vectors reside in large cells. In light of the observations, we design SEIL to share blocks of a large cell (e.g., $cell_{i,j}$) by two lists (e.g., $list_i$ and $list_j$). Such shared blocks enable deduplication on the blocks, thereby saving repeated distance computation for shared blocks. Please note that SEIL can be applied not only to RAIR, but also to any redundant assignment strategy.

Contributions. The contributions of this paper are threefold. First, we propose RAIR, a novel redundant assignment strategy targeting the Euclidean space. We formally prove the effectiveness of the AIR assignment metric. Second, we propose SEIL, a novel list layout to reduce repeated distance computation caused by redundant assignment. Finally, we perform an extensive experimental study to evaluate the benefits of RAIRS. Experimental results show that RAIR out-performs existing redundant assignment strategies, and SEIL can effectively reduce redundant distance computation. Compared to IVF-PQ Fast Scan with refinement (IVFPQfs), RAIRS achieves up to 1.33x speedup in query throughput while maintaining similar recalls across the real-world data sets. We have made the code of RAIRS publicly available: https://github.com/schencoding/rairs.

Outline. The remainder of the paper is organized as follows. Section 2 reviews relevant background and discusses the challenges of redundant assignment. Section 3 overviews the RAIRS solution, then Section 4 and 5 propose RAIR and SEIL, respectively. After that, Section 6 reports the evaluation results. Section 7 discusses related work. Finally, Section 8 concludes the paper.

2 Background

In this section, we review the background on ANNS, IVF, and redundant assignment.

2.1 ANN Search

Problem Definition. Given a set of D-dimension vectors $X = \{x_1, x_2, ..., x_n\}$, where $x_i \in \mathbb{R}^D$ for $i = 1, 2, ..., n$, and a query vector $q \in \mathbb{R}^D$, the k Nearest Neighbors (kNN) problem finds the top-K vectors that are closest to q. In contemporary applications, data sets often reach massive scales with million to billion vectors, and the vectors often consist of hundreds to even thousands of dimensions. The curse of dimensionality [29, 57] makes it impossible to find the exact nearest neighbors without exhaustive searching, which can be prohibitively costly. Consequently, the focus of industry and academia has shifted towards ANNS, sacrificing accuracy slightly for substantial improvement in the processing speed and scalability.

Distance Metrics. The most popular distance metric is the Euclidean distance: $dist(q, x) = \sqrt{\sum_{i=1}^{D}(q^{(i)} - x^{(i)})^2}$, where $x, q \in \mathbb{R}^D$. Other common distance metrics include inner product, cosine similarity, etc. In this paper, our proposed redundant selection method, RAIR, targets the Euclidean distance, whereas the optimized list layout, SEIL, works with all distance metrics.

2.2 IVF (Inverted File Index)

IVF is one of the most widely used ANNS methods in vector databases. In the following, we describe the best-performing IVF variant in practice, IVF-PQ Fast Scan with refinement [3, 5], which we use as the baseline in our work.

- *Product quantization*: PQ [31] is a widely adopted quantization method for accelerating distance computation. It divides the vector dimensions into a number of groups (e.g., with 2 dimensions per group). Every data vector is divided into multiple sub-vectors accordingly. Then, for each dimension group, PQ partitions the sub-vectors into (e.g., 16) clusters (e.g., using K-means). It encodes a vector as its sub-vectors' cluster IDs (a.k.a. code words).

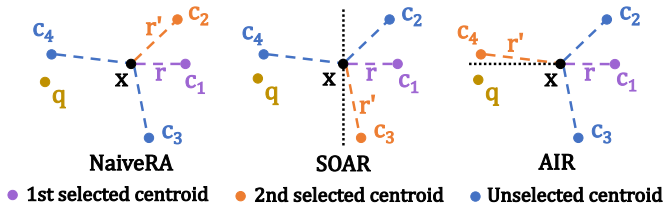


Fig. 2. Different redundant assignment strategies.

IVF-PQ stores the PQ codes along with the vector IDs in the IVF lists. Given a query, IVF-PQ builds LUTs (Look-Up Tables) that contain for each dimension group the squared distance between the query sub-vector and all cluster centroids. To estimate $dist(q, x)$, IVF-PQ looks up the LUT for each code word of x's sub-vector and adds up the squared distances.

- *Refinement*: As estimated distances are not accurate, quantization methods, such as PQ, are often combined with refinement to improve the search quality. The idea is to retrieve a larger number of (e.g., 10×K for a top-K query) vectors from the quantization-enhanced IVF index, then compute the accurate distances and re-rank the retrieved vectors to obtain the top-K results.
- *PQ Fast Scan*: PQ Fast Scan [3] is among a number of existing techniques [3, 4, 11] that exploit SIMD acceleration for distance computation. During index construction, PQ Fast Scan organizes the items (i.e., PQ codes and vector IDs) in an IVF list into packed fixed sized (e.g., 32-item) vector blocks to facilitate SIMD accesses. Then, for query processing, it loads the LUTs into the SIMD registers, and uses SIMD instructions to compute the distances for a block of items at a time.

2.3 Redundant Assignment

Redundant assignment allocates each vector item to multiple IVF lists rather than a single list, thereby decreasing the likelihood of overlooking true top-K nearest neighbors that may originally be assigned to only a list far from the query vector.

NaïveRA (Naïve Redundant Assignment) relies solely on the distance for list selection. SPANN [13] uses NaïveRA to place a subset of vectors in multiple lists in SSD pages. In contrast, we focus on memory-resident environments.

SOAR [50] is a redundant assignment strategy for the inner product distance. Given the first list, it selects the second list with the least $||r'||^2 + \lambda(\frac{r^T r'}{||r||})^2$, while r, r' are the clustering residuals of the first and the second lists, respectively. The second term is non-negative and is minimized when r and r' are close to orthogonal.

Figure 2 illustrates NaïveRA and SOAR. x is a vector. c_1, c_2, c_3, and c_4 are four list centroids. Because c_1 is the nearest centroid, x is first assigned to $list_1$. However, for a query q, x is q's true nearest neighbor, but c_1 is far away from q. Hence, it is likely that the IVF search for q may miss x. We would like to perform redundant assignment for x. As shown in Figure 2, NaïveRA simply selects the second-nearest centroid, c_2. SOAR selects c_3, whose residual vector $r' = x - c_3$ is close to orthogonal to the primary residual vector $r = x - c_1$. Unfortunately, neither c_2 nor c_3 is ideal for q.

In this paper, we propose an AIR strategy optimized for the Euclidean space. To support queries like q, which are closer to the vector but father away from the first assigned list's centroid, AIR selects the second list so that its residual is in the opposite direction of the primary residual. As shown in Figure 2, c_4 is selected, which supports q well.

3 RAIRS Overview

We overview the data structure, the two proposed optimizations, and the index operations of RAIRS.

Data Structure. RAIRS inherits the data structure of IVF-PQ with refinement. As illustrated in Figure 3, it comprises three main components: 1) centroids, 2) inverted lists, and 3) vector data.

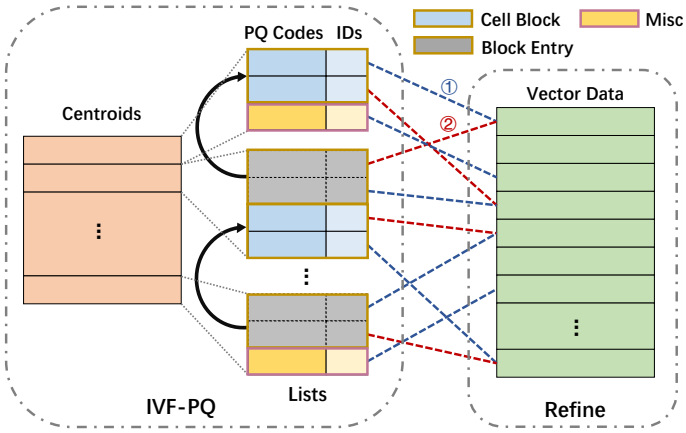


Fig. 3. Overview of the RAIRS index.

The first two components form the IVF-PQ module, while the refine module keeps the original vector data. Each vector is assigned to up to two lists, as depicted by the blue and red dotted lines. The inverted lists store PQ codes and vector IDs, consuming much less memory space than the original vectors. At query time, ANNS first retrieves a set of candidate vectors through IVF-PQ. Then, it accesses the refine module to compute accurate distances for the candidates, and re-ranks the candidates to obtain the final top-K results.

RAIR. Given a query q, IVF traverses the *nprobe* lists whose centroids are the closest to q. However, IVF would miss q's top-K nearest neighbor x if x is not in the closest *nprobe* lists. To address this problem, we propose an AIR (Amplified Inverse Residual) metric for optimized redundant assignment in the Euclidean space. RAIR employs the AIR metric to assign each vector to a second IVF list. In this way, it improves the chance of finding true top-K results given the same *nprobe*. From another angle, it can reduce the number of traversed lists for attaining similar search accuracy, thereby improving query throughput. Moreover, we consider the case where the first and the second selected lists are the same, and we generalize RAIR to multiple assignments. (cf. Section 4)

SEIL. We use the expression "x is in $cell_{i,j}$" to mean that a vector x is assigned to $list_i$ and $list_j$ with redundant assignment. We observe that a subset of the cells contain a large number of vectors. There is strong skew in the number of vectors across the cells. This interesting finding motivates us to optimize the list layout for large cells in order to reduce redundant distance computation.

In the baseline, each list is divided into packed 32-item blocks to facilitate SIMD computation by PQ Fast Scan. A large $cell_{i,j}$ is stored twice in both $list_i$ and $list_j$. If the two lists are both traversed in the same query, distance computation will be performed twice for $cell_{i,j}$, which is wasteful. To address this problem, we propose SEIL that stores the shared blocks of $cell_{i,j}$ only once. In Figure 3, the shared cell blocks are depicted with the blue color. The gray colored block entry points to the shared cell block. For a query, SEIL performs distance computation for the shared blocks only once, decreasing redundant distance computation. (cf. Section 5)

Index Construction. Algorithm 1 shows the procedure for adding a batch of vectors into the RAIRS index. For each vector, the algorithm calls *RairAssign* to assign the vector to two lists (Line 4) and computes its PQ code (Line 6). The original vector is also appended to the vector data to facilitate accurate distance calculations by the refine module (Line 7). Finally, the algorithm invokes *SeilInsert* to insert the vector items into the inverted lists with SEIL layout optimization (Line 8). *RairAssign* and *SeilInsert* will be detailed in Section 4.2 and 5.3, respectively.

Algorithm 1 Add a batch of vectors to RAIRS index.

```
Input: index, vecs, vec_ids
1: function AddVectors(index, vecs, vec_ids)
2:     assignments= []; codes = [];
3:     for (i = 0; i < vecs.len; i ++) do
4:         (listID1, listID2) = RairAssign(index, vecs[i]);
5:         assignments.append( {listID1, listID2, vec_ids[i]} );
6:         codes.append( PQEncoding(vecs[i], index.code_book) );
7:         index.vec_data.append(vecs[i]);
8:     SeilInsert(index, assignments, codes, vec_ids);
9:     index.ntotal += vecs.len;
```

Algorithm 2 ANNS with RAIRS index.

```
Input: index, queries, K, nprobe
Output: results
1: function RairsSearch(index, queries, K, nprobe)
2:     bigK = K * K_FACTOR;
3:     for (i = 0; i < queries.len; i ++) do
4:         LUT = ComputeLookupTable(queries[i], index.code_book);
5:         selected_lists = FindNearestLists(index, queries[i], nprobe);
6:         candidates = SeilSearch(index, LUT, selected_lists, bigK);
7:         results[i] = Refine(index.vec_data, queries[i], candidates, K);
8:     return results;
```

ANNS Query Processing. Algorithm 2 lists the ANNS procedure to find top-K nearest neighbors for a batch of queries. It first computes the number of candidates (*bigK*) to retrieve from the IVF lists as K multiplied by a pre-defined *K_FACTOR* (e.g., 10) (Line 2). For each query vector, the algorithm constructs the PQ distance lookup table (*LUT*) (Line 4) and identifies *nprobe* lists whose centroids are closest to the query (Line 5). Then, it calls *SeilSearch* to traverse each relevant list in the SEIL structure, computes approximate distances based on *LUT* and the PQ codes, and retrieves a set of *bigK* candidates (Line 6). Finally, the refine module attains the top-K results based on exact distance calculations (Line 7). Section 5.3 describes in detail how *SeilSearch* efficiently obtains the desired candidates while reducing redundant distance computation.

Applicability. In this paper, we assume that the vector data can fit into the main memory. We focus on IVF-PQ Fast Scan with refinement as the baseline to demonstrate the effectiveness of our proposed RAIR and SEIL optimizations.

Please note that RAIR can be applied to any IVF-based indices. The redundant assignment strategy is orthogonal to the quantization method, the storage medium, and the hardware optimization of the indices. Moreover, while SEIL is designed to support packed blocks in PQ Fast Scan, the idea of exploiting shared cells to reduce redundant distance computation can be generally applicable. For example, for a disk-resident IVF-flat index, which stores IVF lists on disk and centroids in memory, we can apply the idea of SEIL by replacing the on-disk lists with shared cells of vectors and recording the shared cell addresses along with the list centroids in memory.

4 Redundant List Selection with AIR

We propose AIR (Amplified Inverse Residual) as an optimized list selection metric in the Euclidean space in Section 4.1. Then, we describe and analyze the RAIR algorithm to assign a vector to two lists in Section 4.2. Finally, we consider the generalization of RAIR to assign a vector to multiple lists in Section 4.3.

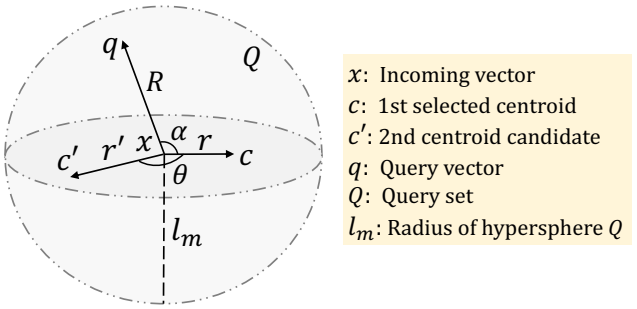


Fig. 4. Geometric relationship of vectors.

4.1 Amplified Inverse Residual

Let x be the data vector to be inserted into the IVF lists and c be the centroid closest to x. We consider queries within a maximal distance l_m of x. Let $Q = \{q : ||q - x|| \leq l_m\}$ be the set of all queries in the hypersphere centered at x with radius l_m. Suppose $q \in Q$ is a random query vector that is uniformly distributed in Q. Figure 4 depicts the geometric relationship of the vectors.

Since c is the closest centroid to x, the list associated with centroid c is the first selected list to assign x. In most cases, representing x with c is satisfactory. However, for some query q, c may not be an ideal representation for x. That is, c lies outside the nearest *nprobe* lists of q, and therefore x is a true top-K nearest neighbor of q but does not appear in the retrieved result with single assignment. In such cases, while q is close to x, q is so far away from c that there are *nprobe* other centroids closer to q than c.

We would like to select the second list for x to accommodate such queries that do not benefit sufficiently from the first selected list with centroid c. As shown in Figure 4, let $\alpha = \angle qxc$. α quantifies how *unhappy* a query q is with c as the first selected centroid for x. This is because in the triangle $\triangle qxc$, larger α indicates longer edge qc, which is $dist(q, c) = ||q - c||$. That is, the larger the α, the farther away q is from c, and the less likely that c appears in the nearest *nprobe* lists of q.

Building on this intuition, we formulate the following loss function for selecting the second list. Let c' denote the centroid of the second selected list. We ensure that the second centroid c' compensates for c among all queries in Q:

$$L(c', c, Q) = E_{q \in Q}[ReLU(-\cos \angle qxc) \cdot (||q - c'||^2 - ||q - x||^2)]$$

In the loss function, the second factor, $||q - c'||^2 - ||q - x||^2$, is easy to understand. It expresses the preference for decreasing the squared Euclidean distance from q to c' compared to q to x. The first factor, $ReLU(-\cos \alpha)$ serves as a weighting term, indicating how important the second centroid c' is to q. When $\alpha \leq \frac{\pi}{2}$, it is likely that x can be visited by q in the list represented by c. In such cases, $-\cos \alpha \leq 0$ and $ReLU$ returns 0. Hence, the contribution of the second factor is ignored, meaning that the second list is not important. On the other hand, if α exceeds $\frac{\pi}{2}$, $ReLU(-\cos \alpha) > 0$. q is close to x but far away from c. We increase the weight for such queries, giving them higher priority during assignment. The larger the α, the higher the weight. Then, we select the second list with the least $L(c', c, Q)$ among all lists.

We prove the following theorem to simplify the computation of the loss function. The full proof is provided in the technical report [64].

Theorem 4.1. *For a set Q of queries that are uniformly distributed in the hypersphere centered at x with radius l_m,*

$$L(c', c, Q) \propto ||r'||^2 + \lambda r^T r'$$

where $r = c - x$, $r' = c' - x$, and $\lambda > 0$ is a constant factor.

Table 1. Comparison of redundant assignment strategies.

Strategy	NaïveRA	SOAR	AIR
Formula	$\|\|r'\|\|^2$	$\|\|r'\|\|^2 + \lambda(\frac{r^T r'}{\|\|r\|\|})^2$	$\|\|r'\|\|^2 + \lambda r^T r'$
Geometric Interpretation	2nd nearest neighbor	Prefer 2nd residual orthogonal to 1st residual	Prefer 2nd residual opposite to 1st residual

Note: r (r') is the clustering residual of the first (second) selected list.

AIR. Based on Theorem 4.1, we set $||r'||^2 + \lambda r^T r'$ as the metric for selecting the second list. From the formula, we see that the metric does not hinge solely on minimizing $||r'||$, the distance from the second selected centroid c' to x. In addition, there is an added penalty term $\lambda r^T r'$. When $||r'||$ is fixed, having r' closer to the inverse of r (i.e., $-r$) leads to a negative second term, which reduces the loss function. This indicates a preference for the second residual r' to be at an inverse direction of the first residual r. Hence, we call this metric AIR (Amplified Inverse Residual) to capture its preference for inverse residuals.

Interestingly, if we set $\lambda = 0$, AIR degenerates to NaïveRA, which selects the second nearest list as the second choice. In our experiments, we set $\lambda = 0.5$ by default. We also perform an in-depth study of the impact of λ on ANNS performance in Section 6.3.

Comparison of Redundant Assignment Strategies. Table 1 compares NaïveRA, SOAR, and AIR. First, NaïveRA aims to minimize the distance of the list centroid for the second selected list. In comparison, both SOAR and AIR consider not only distances but also the relationship between the first and the second selected lists, as evidenced by the second term of their formula. Second, SOAR and AIR are significantly different. The second term in SOAR is proportional to the squared projection of the second residual r' on the first residual r. Thus, SOAR prefers r' to be orthogonal to r, making the second term to close to 0. In contrast, the second term of AIR computes the dot product of the two residuals, and can be negative. AIR prefers r' to be at the inverse direction of r. We compare NaïveRA, SOAR, and AIR experimentally in Section 6.

4.2 RAIR Algorithm

Algorithm 3 shows the *RairAssign* procedure. Given a data vector v, the algorithm first obtains *N_CANDS* lists whose centroids are closest to v (Line 2). The *cand_lists* are sorted in the ascending order of the distance between a list's centroid and v. Then, the AIR loss function is computed for all candidate lists (Line 3–8). After that, the algorithm selects the primary list to assign v as $cand\_lists[0]$, which is the list whose centroid is closest to v (Line 9). It determines the secondary assignment by selecting the list that minimizes the computed AIR loss function (Line 11).

RAIR and Strict RAIR (SRAIR). In cases where the AIR loss function remains minimal for v's primary list, i.e., for any list,

$$||r'||^2 + \lambda r^T r' \geq (1+\lambda)||r||^2,$$

there is little benefit in assigning v to any secondary list. Thus, such vectors are stored only in their first-choice list. This strategy not only curtails space overhead by limiting unnecessary redundancy but also reduces unworthy accesses to vectors when querying, thereby potentially improving overall query performance. We call this strategy RAIR.

In addition to RAIR, we also provide a strict version of RAIR, called SRAIR. SRAIR assigns a vector strictly to two lists. That is, it excludes the first selected list and applies AIR to select the second list from the rest of the lists.

The algorithm uses an *is_strict* flag to support both RAIR and SRAIR (Line 10). When *is_strict* is false, $start$=0 and the second list is selected from all *N_CANDS* lists (Line 10–11). When *is_strict* is true, $start$=1, and $cand\_lists[0]$, which is the first selected list, is excluded from consideration (Line 10–11).

Algorithm 3 Assign a vector to lists using RAIR.

Input: index, v
Output: listID1, listID2

```
1:  function RairAssign(index, v)
2:      cand_lists = FindNearestLists(index, v, N_CANDS);
3:      centroid0 = index.centroids[cand_lists[0]];
4:      residual0 = centroid0 − v;
5:      for (i = 0; i < N_CANDS; i ++) do
6:          centroid = index.centroids[cand_lists[i]];
7:          residual = centroid − v;
8:          loss[i] = L2sqr(residual) + λ· InnerProd(residual0, residual);
9:      listID1 = cand_lists[0];
10:     start = (is_strict) ? 1 : 0;
11:     listID2 = cand_lists[ argmin(loss[start .. N_CANDS-1]) ];
12:     return listID1 < listID2 ? (listID1, listID2) : (listID2, listID1);
```

Reducing Computation Cost with Limited Candidate Lists. In practice, we do not compute the loss function across all *nlist* lists for each vector. Instead, we evaluate only the top N_CANDS nearest lists, as shown in Algorithm 3. Note that this is important for large data sets, where *nlist* is large. *FindNearestLists* can perform an ANNS rather than the exhaustive search, thereby reducing the worst-case $O(nlist \cdot D)$ cost for each vector.

We find that a small N_CANDS (e.g., 10) is often sufficient for the quality of redundant assignment. For instance, in the SIFT1M [30] data set, for over 99.95% of vectors, the minimal loss function is obtained among the top-10 nearest lists when $\lambda = 0.5$ and $nlist = 1024$. We study the setting of N_CANDS in depth in Section 6.3.

Cost Analysis. First, selecting the first list consists of calling *FindNearestLists* (Line 2) and setting $listID1$ (Line 9) in Algorithm 3. Depending on the implementation, the cost of *FindNearestLists* is $O(nlist \cdot D)$ if *FindNearestLists* performs exhaustive search, or can be decreased to $O(sublinear(nlist) \cdot D)$ if *FindNearestLists* performs ANNS. (For example, if *FindNearestLists* performs IVF-based ANNS, the complexity can be $O(\sqrt{nlist} \cdot D)$.)

Second, the remaining Algorithm steps select the second list among N_CANDS candidate vectors using D-dimensional vector computations. Therefore, the cost is $O(N\_CANDS \cdot D)$.

Overall, the time complexity can be expressed as $O((nlist+N\_CANDS)D)$ or $O((sublinear(nlist)+N\_CANDS)D)$ depending on the implementation of *FindNearestLists*.

Note that N_CANDS is often several orders of magnitude smaller than $nlist$. The additional cost of selecting the second list is often much smaller than the cost of selecting the first list. Thus, the cost of list selection in redundant assignment is often close to that of the baseline single assignment.

4.3 Generalization to Multiple Assignments

In the above, RAIR performs two-assignment, assigning each vector to up to two IVF lists. In this subsection, we generalize RAIR to m-assignment, where $m \geq 3$.

Given $(m-1)$ selected lists, we select the m-th centroid by considering the losses with regard to all prior selected centroids:

$$L_m(c', c_1, ..., c_{m-1}, Q) = \underset{1 \leq i \leq m-1}{aggr} L(c', c_i, Q) \propto ||r'||^2 + \lambda \underset{i}{aggr}\ r_i^T r'$$

For $aggr$, we evaluate three functions (i.e., max, min, and avg) in Section 6.3. Our results show that max performs the best.

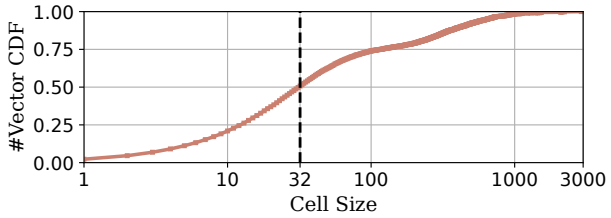


Fig. 5. Characteristics of cells after redundant assignment.

Note that assigning vectors to more than two lists does not necessarily improve ANNS performance. Distributing each vector across additional lists can reduce the required number of traversed lists (*nprobe*) for queries. However, the average IVF list size grows, incurring larger number of distance computing operations. Our experiments show that two-assignment yields the smallest number of distance computations.

5 Shared-Cell Enhanced IVF Lists

We discuss the problems of the baseline list layout for supporting redundant assignment in Section 5.1. Next, we propose the SEIL layout optimization in Section 5.2. Finally, we describe the algorithms for the SEIL-enhanced index in Section 5.3.

5.1 Challenges of Baseline List Layout

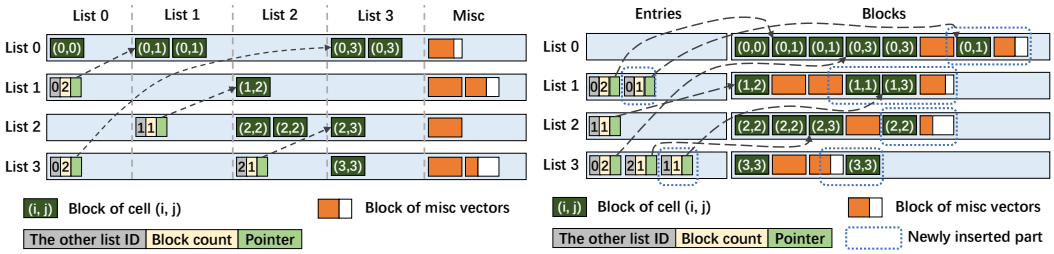
Consider the case where vector x is assigned to $list_i$ and $list_j$. In the baseline list layout, the vector item (including x's PQ code and its vector ID) is stored in both $list_i$ and $list_j$. PQ Fast Scan further packs every 32 vector items into a block to facilitate SIMD acceleration in each list. This baseline list layout suffers from two issues: 1) redundant distance computations at query time if both $list_i$ and $list_j$ are among the *nprobe* lists chosen for a query, and 2) increased space cost for storing the vector item twice.

The first issue lowers the query throughput. One naïve solution is to collect all vector IDs from the *nprobe* chosen lists, deduplicate the vector IDs, then perform distance computation. Another solution is to build a hash table to keep track of the vector IDs whose distances have been computed, then check the hash table on the fly to avoid any redundant computation. However, both solutions require retrieving the vector IDs before distance computation. Unfortunately, this requires unpacking the packed blocks, which would break the SIMD computation steps and drastically reduce the benefit of PQ Fast Scan. Moreover, both solutions employ either sorting or hashing based algorithms for all vector items in *nprobe* lists, which can lead to non-trivial additional time and space cost.

The second issue is mitigated by duplicating the vector items rather than the D-dimensional vectors in the lists. Moreover, if the first and the second assigned lists are the same, RAIR avoids to store the vector item twice. Nevertheless, it would be nice to further reduce the space cost introduced by redundant assignment.

5.2 SEIL Layout

Characteristics of Cells. We study the characteristics of the cells in Figure 5. Recall that $cell_{i,j}$ contains all vectors that are assigned to both $list_i$ and $list_j$. Since $cell_{i,j}$ and $cell_{j,i}$ are essentially the same, we ensure $i \leq j$ for $cell_{i,j}$. For a vector that is assigned to only a single list i in RAIR, we set its cell to $cell_{i,i}$. We count the vectors in each cell after redundant assignment. The x-axis shows the cell size (i.e., the number of vectors in a cell) in the logarithmic scale. The y-axis reports the Cumulative Distribution Function (CDF) of how vectors are distributed across cells for the SIFT1M data set. 1.0 corresponds to the sum of all cell sizes.



(a) Illustration of shared cells (b) List structures after two batch of insertions

Fig. 6. SEIL list layout.

In Figure 5, we draw a dotted line at cell size = 32. Since a block contains 32 vector items in PQ Fast Scan, we consider a cell to be large if its cell size ≥ 32. In the figure, large cells are to the right of the dotted line. From the figure, we observe that 1) large cells contain about 50% of all vectors, and 2) there exist very large cells that contain hundreds to thousands of vectors.

This observation of a high degree of concentration of vectors in large cells motivates us to exploit shared cells. Our idea is to share the vectors of a large $cell_{i,j}$ in both $list_i$ and $list_j$ as much as possible for reducing redundant distance computations at query time and decreasing the space cost due to redundant assignment.

SEIL List Structure. The SEIL list structure is depicted in Figure 6. Figure 6(a) illustrates the idea of shared cells. $cell_{i,j}$ is drawn at row i column j. Vectors in cells are grouped into 32-item blocks. Suppose a cell contains $nitems$ vectors. Then, we generate ($nitems$/32) blocks. The remaining ($nitems$%32) vectors are appended to blocks in the miscellaneous area. The blocks of $cell_{i,j}$ are stored only once in memory. They are shared by $list_i$ and $list_j$. For example, $cell_{0,1}$'s blocks are shared by $list_0$ and $list_1$. The blocks are physically stored in $list_0$. $list_1$ contains a (the other list ID, block count, pointer) entry that references the shared blocks in $list_0$.

Figure 6(b) shows the physical list structures. Each list contains an array of reference entries and an array of 32-item blocks. Shared blocks of $cell_{i,j}$ ($i < j$) are physically stored in $list_i$, while $list_j$ stores the reference entry/entries pointing to the physical blocks. The figure also depicts the structures after two batches of insertions. The entries and blocks of the second batch are highlighted with dotted rounded rectangles. They are appended to the end of the entry arrays and block arrays.

We would like to point out several design considerations. First, a reference entry can point to multiple blocks of the same cell. For a batch of insertions, SEIL stores multiple blocks of the same cell contiguously. A reference entry points to the first block with the other list ID and pointer field. The block count is the number of contiguous blocks in the cell. Second, for the new batch of vectors, a new entry referencing the same other list can be generated. For example, for the second batch, $cell_{0,1}$ sees a new block. Then, the new block is stored in $list_0$, and a new reference entry pointing to $list_0$'s new block is appended to $list_1$. Now, $list_1$ contains two reference entries both pointing to $list_0$. Third, the last miscellaneous block of a list may contain less than 32 vector items, which is depicted as half-filled orange rectangles. All other blocks are full. For a new batch, we fill the last miscellaneous block of the previous batch before generating new miscellaneous blocks. Finally, for a vector item stored in the miscellaneous blocks, we embed its other assigned list ID in the unused high-order bits of the vector ID. Suppose x is one of the remaining ($nitems$%32) vectors in $cell_{i,j}$. Then, we embed j (i) when storing x in a $list_i$'s ($list_j$'s) miscellaneous block. This simplifies the deduplication of miscellaneous vectors.

SEIL-Optimized Deduplication. For blocks in shared cells, SEIL enables cell-level deduplication to reduce redundant distance computation. The basic idea is to determine if a shared cell has been

processed in the same query, and skip the SIMD distance computation for all blocks of the shared cell if the cell has been seen.

To achieve this, we distinguish physical blocks from reference entries. For physically stored blocks of shared cells, we always perform distance computation. For reference entries, we need to determine whether their corresponding blocks have been accessed before in the same query. This is accomplished with a $listVisited$ hash table that keeps track of the visited lists in the query. $listVisited$ is probed with the other list ID of a reference entry. If it exists, then the corresponding blocks have already been processed and thus the reference entry is skipped. Otherwise, distance computation is carried out for the blocks represented by the reference entry.

For vectors in the miscellaneous area, we cannot avoid distance computation because of the packed blocks of PQ Fast Scan. However, it is still necessary to deduplicate the vectors after the distance computation so that the same vector won't appear twice in the returned result. Instead of maintaining detailed records of all miscellaneous vectors accessed, we exploit the above $listVisited$ hash table to simplify the deduplication. Basically, for each miscellaneous vector x, we check whether the other list ID embedded in x's vector ID has already been accessed in $listVisited$, and immediately skip the vector if the check returns true.

5.3 SEIL Algorithms

Constructing SEIL-Optimized Lists. Algorithm 4 shows the procedure to insert a batch of vectors into SEIL-optimized lists. The algorithm sorts the assignment items in ascending order so that the items in the same cell are contiguous in the *assigns* array (Line 8). Then, the algorithm scans the *assigns* array twice in two for-loops.

The first for-loop computes the number of shared blocks and the number of items in the miscellaneous area for each list (Line 9–15). Each loop iteration processes the items in the same cell. It obtains the cell (Line 10), counts the number of items in the cell (Line 11), computes the number of blocks and the remaining items (Line 12), and updates the relevant statistics (Line 13–15). After that, this information is used to allocate space for the lists (Line 16).

The second for-loop populates the lists with the items (Line 17–32). It obtains *cell*, *nitems*, *nblocks*, and *nmisc* in the same way as in the first for-loop (Line 18–20), then appends shared blocks (Line 23–25) and miscellaneous items (Line 26–27) in the first list. If the second list is not the same as the first list, the algorithm also appends the miscellaneous items in the second list (Line 31–32), and creates a reference entry to point to the shared blocks of the first list (Line 30). Vectors within the same block have their PQ codes arranged in the PQ Fast Scan format.

Searching SEIL-Optimized Lists. Algorithm 5 uses two structures to facilitate the search: $rqueue$ that maintains the top-$bigK$ vectors, and $listVisited$ that keeps track of visited lists for deduplication purposes. The algorithm initializes the two structures (Line 2–3), then goes into a loop to search each selected list (Line 4–18).

For each list, the algorithm processes the reference entries (Line 6–10), shared blocks stored in the current list (Line 11–13), and blocks in the miscellaneous area (Line 14–17). For the reference entries, cell-level deduplication is used to check if all blocks pointed to by an entry can be skipped (Line 7). For the physically stored shared blocks, it is guaranteed that there is no duplication since the associated reference entry will be checked in the other list. For the miscellaneous area, the vectors are deduplicated after distance computation using $listVisited$ (Line 16).

PQ Fast Scan is invoked for each block to efficiently compute the distances with SIMD instructions (Line 9, 12, 15). At the end of each loop iteration, the current list is added to $listVisited$ (Line 18). Finally, the algorithm returns the top-$bigK$ candidates (Line 19).

Implementation: Cache Optimization for Query Batch. ANNS can be invoked for a batch of queries, which is standard in various ANNS benchmarks [5, 70]. Let each (query, selected list) be a

Algorithm 4 Insert a batch of vectors to SEIL-optimized lists.

```
Input: index, assigns, codes, vec_ids
function GetCell(assign)
    return (assign.listID1, assign.listID2);
function CountItemsWSameCell(assigns, i, cell)
    for (j = i+1; j < assigns.len; j++) do
        if (GetCell(assigns[j]) != cell) then break;
    return j − i;
function SeilInsert(index, assigns, codes, vec_ids)
    Sort assigns in ascending order of {listID1, listID2, vec_id};
    for (i = 0; i < assigns.len; i += nitems) do
        cell = GetCell(assigns[i]);
        nitems = CountItemsWSameCell(assigns, i, cell);
        nblocks = nitems / BLK_SZ; nmisc = nitems % BLK_SZ;
        list_nb[cell.listID1] += nblocks; list_nm[cell.listID1] += nmisc;
        if (cell.listID1 != cell.listID2) then then list_nm[cell.listID2] += nmisc;
    Allocate space according to list_nb and list_nm;
    for (i = 0; i < assigns.len; i += nitems) do
        cell = GetCell(assigns[i]);
        nitems = CountItemsWSameCell(assigns, i, cell);
        nblocks = nitems / BLK_SZ; nmisc = nitems % BLK_SZ;
        list1 = index.lists[cell.listID1];
        bptr = list1.getNextBlockPointerInSharedCell();
        for (b = 0; b < nblocks; b++) do
            begin = i + b*BLK_SZ; end = begin + BLK_SZ − 1;
            list1.appendBlockInSharedCell(assigns, begin, end);
        for (j = i + nblocks*BLK_SZ; j < i+nitems; j++) do
            list1.appendItemInMiscArea(assigns, j);
        if (cell.listID1 != cell.listID2) then
            list2 = index.lists[cell.listID2];
            list2.appendReferenceEntry(cell.listID1, nblocks, bptr);
            for (j = i + nblocks*BLK_SZ; j < i+nitems; j++) do
                list2.appendItemInMiscArea(assigns, j);
```

computation task. One implementation is to group the tasks by queries, and process all the tasks of the same query before moving on to the next query. However, this can be suboptimal. In many cases, a list is visited by multiple queries in a query batch. Since the data accessed by a query is often much larger than the CPU cache, the implementation incurs a lot of CPU cache misses for accessing the same list in multiple queries.

To deal with this problem, our implementation employs an optimization technique to improve the CPU cache performance, which is available in Milvus [52] and Faiss [18]. The idea is to group the tasks by lists, and process all the queries for the same list back to back. In this way, a list stays in the CPU cache for all but the first query searching it. For conciseness of presentation, we have omitted the pseudo-code of this optimization in Algorithm 5.

Cost Analysis. We consider the cost of *SeilInsert* in Algorithm 4. Suppose n vectors are to be inserted. For sorting *assigns*, we can employ the bucket sort with *nlist* buckets in two passes, which takes $O(n)$ time. In the two for-loops, *CountItemsWSameCell* visits each item in the *assigns* array, resulting in $O(n)$ cost. Packing the n vector items with their PQ codes and vector IDs into blocks in *appendBlockInSharedCell* and *appendItemInMiscArea* takes $O(n \cdot D)$ cost. The rest of

Algorithm 5 Searching SEIL-optimized lists.

```
Input: index, LUTs, selected_lists, bigK
Output: candidates
function SeilSearch(index, LUT, selected_lists, bigK)
    rqueue.init(bigK);
    listVisited.init();
    for (each listID in selected_lists) do
        list = index.lists[listID];
        for (each (otherLID, nblocks, bptr) in list.ref_entries) do        ▷ process reference entries
            if (! listVisited.exist(otherLID)) then
                for (b = 0; b < nblocks; b ++) do
                    results = PQFastScan(LUT, bptr[b]);
                    rqueue.update(results);
        for (each block in list.shared_cell_blocks) do   ▷ process shared blocks w/o duplication checking
            results = PQFastScan(LUT, block);
            rqueue.update(results);
        for (each block in list.misc_blocks) do                  ▷ process items in miscellaneous area
            results = PQFastScan(LUT, block);
            results = RemoveVectorIfVisited(listVisited, results);
            rqueue.update(results);
        listVisited.add(listID);
    return rqueue.output(bigK);
```

the operations are performed for each cell. Since the number of cells is bounded by n, their cost is bounded by $O(n)$. As a result, the cost of *SeilInsert* is $O(n \cdot D)$, which is dominated by packing vector items into blocks. Our SEIL optimization avoids redundant storage of blocks for shared cells, thereby reducing the constant factor of this cost.

Next, we consider the cost of $AddVector$ in Algorithm 1, which performs the complete insertion procedure, involving both RAIR and SEIL. As described in Section 4.2, the cost of *RairAssign* for each vector is $O((sublinear(nlist) + N\_CANDS)D)$. For n vectors, its cost is $O((sublinear(nlist) + N\_CANDS)nD)$. The remaining operations in the for-loop of $AddVector$, including PQ encoding for vectors, take $O(n \cdot D)$ time. *SeilInsert* takes $O(n \cdot D)$ time. Therefore, the overall cost is $O((sublinear(nlist) + N\_CANDS)nD)$.

On the SIFT1M data set, inserting all data vectors to the RAIRS index takes 15.8s, while the single-assignment IVFPQfs baseline requires 14.0s and the HNSW index requires 136.0s. The training time is 13.3s for both RAIRS and IVFPQfs. Therefore, the index construction time is 29.1s and 27.3s for RAIRS and IVFPQfs, respectively. Although RAIRS incurs a 6.6% slowdown relative to IVFPQfs in index construction, it is still much faster than HNSW. Consequently, we consider the additional construction overhead introduced by RAIRS to be within practical bounds. Additional results on insertions are provided in Section 6.2.

For *SeilSearch* in Algorithm 5, let $n_{vec\_selected}$ be the total number of vectors in the selected lists. Then, the worst-case cost of *SeilSearch* is $O(n_{vec\_selected}D)$. Suppose n_{vec_shared} is the number of vectors in the blocks of cells shared by two selected lists. Then, the SEIL-optimized cell-level deduplication reduces the cost to $O((n_{vec\_selected} - n_{vec\_shared})D)$.

6 Evaluation

We start by describing the experimental setup in Section 6.1. Next, we report the overall performance of RAIRS in Section 6.2, followed by in-depth analysis of individual techniques and algorithm

Table 2. Data sets used in the experiments.

Data Set	Distance	#Dim	#Item	#Query	Size
SIFT1M [30]	Euclidean	128	1,000,000	10,000	488MB
SIFT1B [30]	Euclidean	128	1,000,000,000	10,000	477GB
MSong [36]	Euclidean	420	994,185	1,000	1.56GB
GIST [30]	Euclidean	960	1,000,000	1,000	3.58GB
OpenAI [70]	Euclidean	1536	5,000,000	1,000	28.61GB
T2I [8]	Inner product	200	10,000,000	100,000	7.45GB

parameters in Section 6.3. Finally, we study the flexibility of the SEIL optimized list layout by applying it to SOAR in Section 6.4.

6.1 Experimental Setup

Machine Configuration. We conduct all experiments on a server equipped with an Intel(R) Xeon(R) Platinum 8360Y CPU (2.40GHz, 36 cores, 64KB L1 cache per core, 1.25MB L2 cache per core, 54MB shared L3 cache) and 1TB 3200MT/s DDR4 memory, running CentOS 7.9.2009. The CPU supports both AVX2 and AVX-512 SIMD instructions[1]. C/C++ code is compiled using GCC 9.3.1 with -O3.

Implementation. We implement the RAIRS index based on Faiss v1.8.0 [18] and employ OpenBLAS v0.3.3 for linear algebra support. RAIRS is implemented as a number of subclasses of the IndexIVF class of Faiss. It interacts with Faiss through Faiss's public APIs. We write approximately 2,200 lines of code.

In the experiments, both RAIR assignment and SEIL query routines are conducted with batches of vectors by default. For index construction with $AddVectors$, the set of all data vectors are provided in a batch to build the RAIRS index, which performs RAIR assignment and constructs SEIL-optimized lists. For query processing, each thread processes a batch of query vectors in parallel to maximize throughput. We perform a bulk execution of the $FindNearestLists$ function, retrieving the nearest lists for all query vectors in the query batch. Then, we employ the cache optimization for query batches as described in Section 5.3. We scan each list for all associated queries, which allows the current list to remain in the CPU cache, thereby improving overall performance.

To support deletion, the FAISS IVF index maintains a map from each vector ID to the vector's list ID and in-list position. In RAIRS, we modify the map to include up to 2 list IDs and in-list positions for each vector ID. Given a vector ID to be deleted, if the in-list position is in a shared-block, we set the corresponding ID in the packed block to an invalid ID. If the position is in a misc-block, we replace this entry with the last vector ID in misc-blocks. Since each vector is assigned up to two lists, RAIRS modifies up to twice as many entries compared to the baseline IVFPQfs for a deletion.

Solutions to Compare. We compare the following popular ANNS indexing methods and redundant assignment strategies:

- **IVF** [47]: Class IVFFlat in Faiss 1.8.0. This is the plain IVF index, whose list traversal performs accurate distance computation. We use the same IVF parameters as RAIRS in each data set.
- **HNSW** [39]: Class HNSW in Faiss 1.8.0. This is a widely-used graph-based ANNS index. We follow the default settings in its original work [39] (e.g., $efConstruction = 500$) except for M. We set $M = 32$, which is the default setting of Faiss, because it achieves better performance than $M = 16$ of the original work.

[1]The IndexIVFPQFastScan class in FAISS v1.8.0 uses AVX2 256-bit SIMD instructions, but the compilation of FAISS exploits AVX-512 instructions to generate libfaiss_avx512.so. Our experiments use libfaiss_avx512.so. In our experiments, downclocking does not occur.

- **IVFPQfs** [3]: The IVFPQFastScan in Faiss 1.8.0 implements IVF-PQ Fast Scan. A refine layer is added to improve the recalls. The main distinctions between IVFPQfs and RAIRS are the RAIR assignment and the SEIL layout. IVFPQfs performs the baseline single assignment. We use the same parameters as RAIRS.
- **NaïveRA** [13]: We implement the naïve strategy of redundant assignment. It uses the IVF-PQ Fast Scan with Refine structure and the same parameters as RAIRS. SEIL is not enabled by default.
- **SOARL2** [50]: We replace the redundant assignment strategy of NaïveRA with SOAR. Please note that SOAR is originally designed for the inner product distance. Here, we directly apply SOAR to the L2 distance in the Euclidean space.
- **RAIR and SRAIR**: RAIR and Strict RAIR (cf. Sec 4.2) without the SEIL list layout optimization.
- **RAIRS and SRAIRS**: RAIR and Strict RAIR with the SEIL list layout optimization.

Thread-level parallelism is handled by the Faiss library. We ensure that all solutions run with the same number of threads and the same parallelization scheme. Hyper-threading is not used. For every index with the IVF-PQ Fast Scan + Refine structure, we consistently employ the cache optimization for query batches.

Data Sets. We use the following representative real-world data sets covering a diverse range of vector dimensions, vector counts, and application scenarios in the experiments. The features of the data sets are summarized in Table 2.

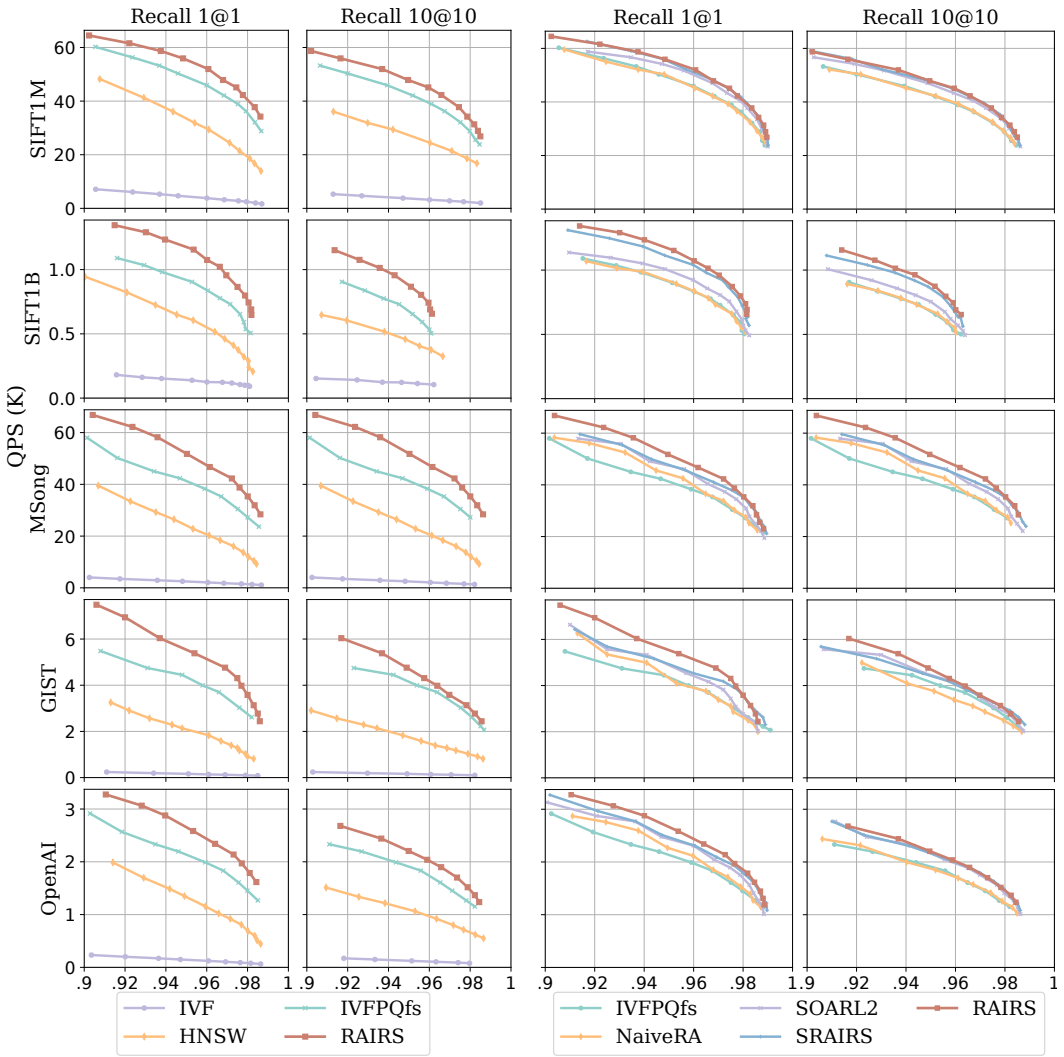
- **SIFT** [30]: SIFT (Scale-Invariant Feature Transform) descriptors are derived from an image data set. We use two SIFT data sets, i.e., SIFT1M and SIFT1B, which contain 1 million and 1 billion vectors, respectively. In SIFT1B, the original descriptors are stored as 8-bit integers; for consistency with the other data sets, we convert them to 32-bit floats during pre-processing.
- **MSong** [36]: The million song data set contains features for a million popular music tracks.
- **GIST** [30]: GIST descriptors are generated from an image data set, which capture the spatial structure of scenes.
- **OpenAI**: This OpenAI embedding data set is generated from an open sourced C4 data set from the Common Crawl data. We download it using the command line tool of VectorDBBench [70] with L2 distance as the metric type.
- **T2I** [8]: The Yandex Text-to-Image data set contains image embeddings as data vectors and text embeddings as queries.

Since RAIRS targets the Euclidean space, our main experiments use SIFT, MSong, GIST, and OpenAI. Unlike the other data sets, the distance metric of T2I is inner product. We use T2I to study the applicability of SEIL to the original SOAR in Section 6.4.

Parameter Setting. By default, we set $nlist = 1024$ except for data set OpenAI [70] ($nlist = 2048$), T2I [8] ($nlist = 3172$), and SIFT1B [30]($nlist = 32768$). These $nlist$ values are close to $O(\sqrt{\#Item})$ of each data set, as suggested in the Faiss library [17]. For PQ Encoding, we set the number of sub-groups $M_{PQ}=\frac{\#Dim}{2}$, and the bit length of the code word for each sub-group $nbits_{PQ} = 4$. In the refine module, we set $K\_FACTOR = 10$ for top-1 and top-10 queries. (10 is a common setting used for faiss-ivfpqfs experiments in ANN-Benchmark results [5].) For top-100 queries, we set $K\_FACTOR = 4$ to balance the time for list traversal and vector refinement. For RAIRS, we set $\lambda = 0.5$ and $N\_CANDS = 10$ based on our experimental study of parameter settings in Section 6.3.

At query time, we vary the search parameter (e.g., *nprobe* in IVF-based indices and $efSearch$ in HNSW) to achieve different trade-offs between query speed and search quality, which contribute to the points of the reported curves in the experimental results.

Performance Metrics. 1) **Recall-QPS**. Since ANNS inherently involves a trade-off between accuracy and speed, its performance cannot be fully captured by throughput alone. Hence, we report



(a) Comparison with popular ANNS methods (b) Varying assignment strategies

Fig. 7. Overall ANNS performance.

Table 3. Percentage of vectors whose 2nd-choice centroid under SOARL2 matches that under AIR.

	SIFT1M	**SIFT1B**	**MSong**	**GIST**	**OpenAI**
SOARL2	95.14%	72.10%	93.93%	91.55%	93.40%

recall-QPS curves, which reflect the balance between retrieval quality and processing efficiency. For top-K queries, the recall $k@K$ is the average percentage of true top-K nearest neighbors in the query result. The query throughput is reported as QPS (Queries Per Second). 2) **Recall-DCO**. The main cost of ANNS query processing is distance computation [21]. DCO is the number of Distance Computing Operations per query. We use DCO to understand the effectiveness of redundant assignment strategies. Similar to recall-QPS curves, a recall-DCO curve is constructed by plotting the recall against the corresponding DCO while varying the search parameter (e.g., *nprobe*).

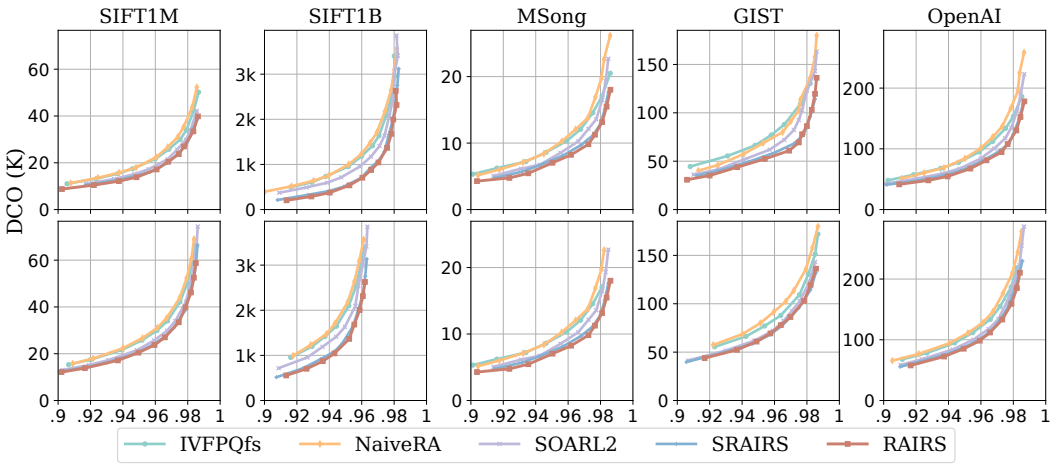


Fig. 8. DCO per query under various assignment strategies. The top row: Recall 1@1-DCO; The bottom row: Recall 10@10-DCO.

6.2 Overall Performance

Comparison with Popular ANNS Methods. Figure 7a shows the Recall-QPS curves of IVF, HNSW, IVFPQfs, and RAIRS on five representative real-world data sets. Each row of sub-figures correspond to a data set. The left column reports results for top-1 queries, while the right column displays performance for top-10 queries. In each plot, the closer to the top-right corner, the better performance. For each search parameter and the resulting recall, we run the experiments for 10 times and report the average QPS.

As shown in Figure 7a, IVFPQfs and RAIRS achieve significantly higher performance than IVF and HNSW. This performance advantage comes mainly from PQ Fast Scan [3]'s SIMD acceleration, which processes packed blocks in the list traversal. This approach is substantially faster than distance computation for each individual vector. In addition, the refine layer compensates for the decrease of accuracy caused by the PQ encoding.

Among the ANNS methods, our proposed RAIRS achieves the best performance. It combines the RAIR redundant assignment and the SEIL optimized list layout to improve the IVF-based ANNS performance. Compared to the second best performing method (i.e., IVFPQfs), RAIRS achieves up to 1.33x speedup in query throughput with similar recalls across all the real-world data sets.

Comparison of Various Assignment Strategies. Figure 7b shows the Recall-QPS curves of five assignment strategies for top-1 and top-10 queries on the five real-world data sets. The five solutions are all based on IVF-PQ Fast Scan with refinement. From Figure 7b, we make the following observations. (1) Compared to the baseline single assignment (i.e., IVFPQfs), redundant assignment with NaïveRA is not better, especially at high recalls. NaïveRA is actually worse than IVFPQfs for top-10 queries on GIST. This indicates the importance of an optimized list selection strategy. (2) Among all redundant assignment strategies, RAIRS achieves the best performance across all the data sets. At 0.95 recall, RAIRS achieves throughput improvement of 1.07–1.33x, 1.11–1.32x, and 1.01–1.23x compared to IVFPQfs, NaïveRA, and SOARL2, respectively. (3) SRAIRS with strict two assignments per vector are comparable to RAIRS on SIFT1M and SIFT1B, but worse than RAIRS on MSong, GIST, and OpenAI. This means that when the first and second chosen lists are the same, it is better to store the vector item only once. (4) RAIRS is significantly better than SOARL2 in most cases because unlike SOAR, RAIRS is optimized for the Euclidean space. For SIFT1M at recall 1@1, and SIFT1M and OpenAI at recall 10@10, SOARL2 exhibits performance comparable to RAIRS. AIR

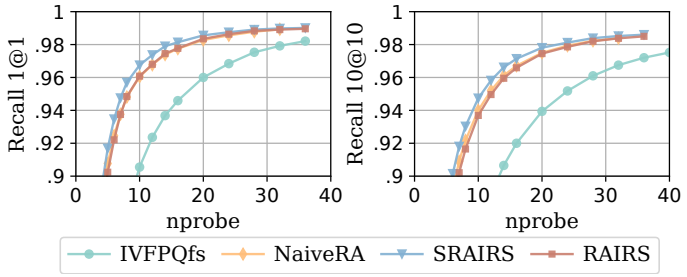


Fig. 9. Recalls varying *nprobe* on SIFT1M.

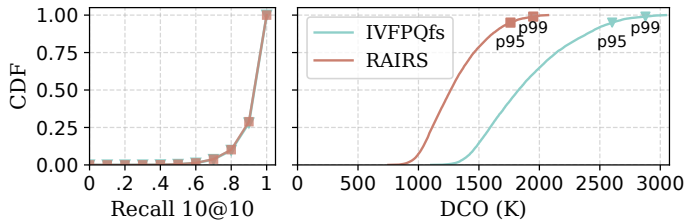


Fig. 10. Distribution of recalls and DCOs on SIFT1B.

prefers to select a c' such that $r' = c' - x$ is closer to the inverse of $r = c - x$ (i.e., the closer θ to 180 degrees, the better). However, depending on the given data set and the vector in consideration, the angle θ for the actual c' may be much smaller than 180 degrees, resulting in similar assignments as in SOARL2. Table 3 reports the percentage of vectors whose 2nd-choice centroid under SOARL2 matches the 2nd-choice under AIR. We see that AIR and SOARL2 chooses the same assignment for 72.10%–95.14% vectors. For the above cases where SOARL2 and RAIRS have similar performance (i.e., SIFT1M and OpenAI), there are high percentages of 2nd-choice matches. From another angle, the 4.86%–27.90% different assignments lead to the performance benefit of RAIRS over SOARL2. Generally, the larger the difference, the larger the potential benefit of RAIRS.

Understanding Assignment Strategies with DCO. Figure 8 shows the Recall-DCO curves of various assignment strategies. The closer to the bottom-right corner, the better. The Recall-DCO curves display similar trends as the Recall-QPS curves in Figure 7b. Since the solutions are all based on IVF-PQ Fast Scan with refinement, the difference in query throughput is primarily due to difference in the number of computed distances. At 0.95 recall, RAIRS reduces the DCO by factors of 0.64–0.83x, 0.62–0.78x, and 0.73–0.99x compared to IVFPQfs, NaïveRA, and SOARL2, respectively.

Note that the benefit of RAIRS comes not only from higher recalls but also from lower DCOs, as evidenced by Figure 8. RAIRS can achieve better recalls with the same DCOs, and the same recalls with lower DCOs (which result from lower *nprobe*'s). Figure 9 plots recalls while varying *nprobe*. Comparing the *nprobe*'s for achieving similar recalls, we see that the *nprobe*'s of SRAIRS and RAIRS are 42.3%–46.5% and 48.1%–53.1% of the setting of the baseline single assignment IVFPQfs, respectively. For RAIRS, since a faction of vectors are assigned only once, achieving recalls comparable to SRAIRS can require a slightly larger *nprobe*.

Given the search parameter, DCO is deterministic for given queries on a given ANNS index. In contrast, QPS is less stable due to run-time variations. Therefore, for solutions based on IVF-PQ Fast Scan with refinement, we show the ANNS performance in DCO in the rest of the evaluation.

Distribution of Recalls and DCOs. For the SIFT1B + 0.95 recall experiment in Figure 8, we compute the recall and the DCO for each of the 10,000 queries in the top-10 search experiment. We plot the CDF (Cumulative Distribution Function) of recalls and DCOs in Figure 10. From the

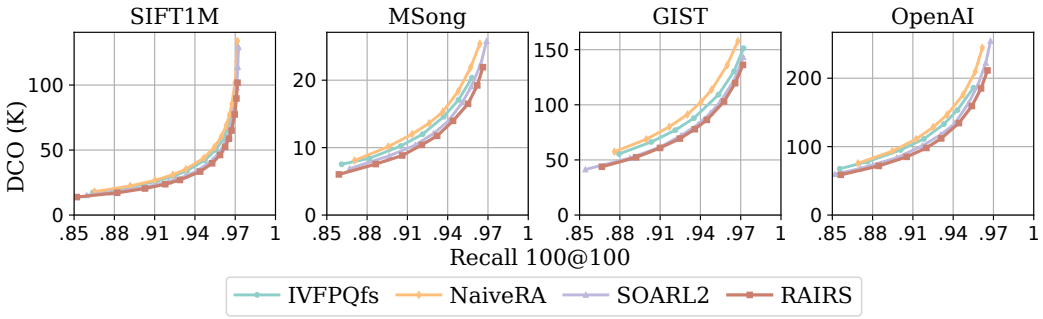


Fig. 11. Performance for top-100 queries.

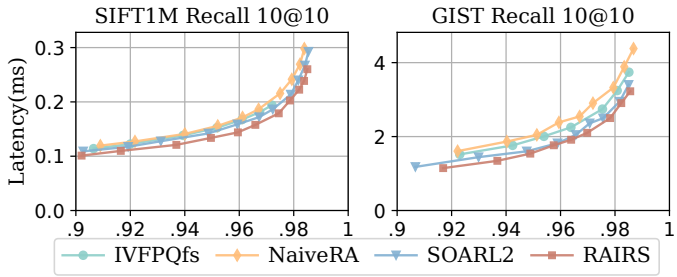


Fig. 12. Performance for one query at a time.

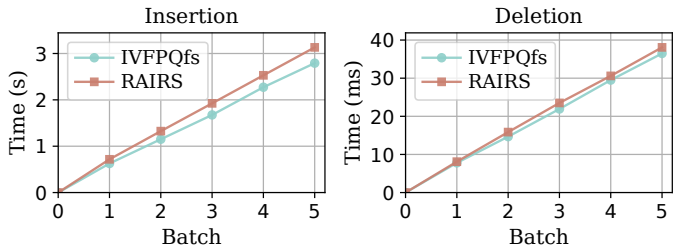


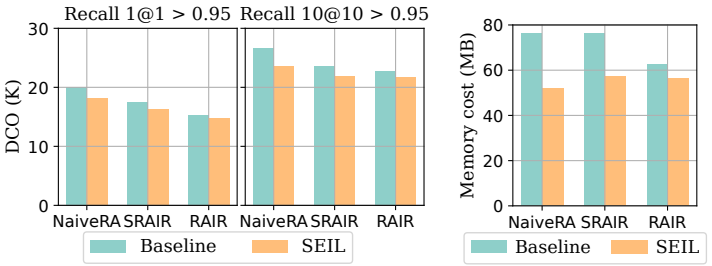
Fig. 13. Performance of insertions and deletions.

figure, we see that the recall CDF curves almost overlap, but RAIRS's DCO curve is clearly to the left of IVFPQfs's DCO curve. This means that compared to IVFPQfs, RAIRS significantly reduces the DCOs for almost all queries while achieving similar recalls. Moreover, the recall variance is low; over 89% of queries achieve 0.8–1.0 recalls. The p99 DCOs of RAIRS is 1.50x of the average. The DCO variance across queries is moderate.

Performance for Top-100 Queries. Figure 11 compares the Recall-DCO curves of IVFPQfs, NaïveRA, SOARL2, and RAIRS for top-100 queries. We see that RAIRS achieves the best performance across all the data sets, which is consistent with the results for top-1 and top-10 queries in Figure 8.

Performance for One-Query-at-a-Time. In this experiment, we run one query at a time to understand the single-threaded query latency without the cache optimization for query batches. Figure 12 shows the query latency-recall curves for SIFT1M and GIST1M. We see that RAIRS achieves the lowest latency among all single assignment and 2-assignment strategies. The benefit of RAIRS is similar to that seen in the batched-query scenarios.

Performance of Insertions and Deletions. Figure 13 shows the performance of vector insertions and deletions for RAIRS and IVFPQfs on SIFT1M. For insertions, we construct the index with 800,000 vectors, then perform five insertion batches, each inserting 40,000 new vectors. For deletions, we build the index with the full data set, then execute five batches of deletions, each removing 40,000



(a) ANNS performance (b) Memory cost

Fig. 14. Ablation study for RAIR and SEIL on SIFT1M.

Table 4. Memory cost for data sets with Euclidean distance.

Data Set	IVFPQfs	NaïveRA	NaïveRA+SEIL	RAIR	RAIRS
SIFT1M	38.2 MB	76.4 MB	52.0 MB	62.5 MB	56.25 MB
SIFT1B	37.3 GB	74.6 GB	42.9 GB	69.7 GB	43.3 GB
MSong	107.2 MB	214.5 MB	145.0 MB	162.8 MB	152.4 MB
GIST	240.6 MB	478.5 MB	325.0 MB	403.3 MB	358.7 MB
OpenAI	1.83 GB	3.66 GB	2.39 GB	3.15 GB	2.53 GB

vectors. Compared to the baseline IVFPQfs, RAIRS's insertion and deletion throughput is 12.2% and 4.4% lower. This is because each vector is assigned up to two lists and RAIRS modifies up to twice as many entries as IVFPQfs. Please note that ANNS is typically much more frequent than vector insertion and deletion operations. While incurring slight extra cost for insertions and deletions, RAIRS achieves higher ANNS query performance.

6.3 In-depth Analysis

Ablation Study for RAIR and SEIL. Figure 14a compares the DCO of NaïveRA, SRAIR, and RAIR with and without the SEIL list layout optimization on the SIFT1M data set. For each compared solution, we report the DCO for the setting where the query recall just exceeds 95%. In the figures, the lower the DCO, the better.

We see that RAIR out-performs NaïveRA with or without SEIL. Without SEIL, RAIR cuts down DCO by 24.0% and 14.9% compared to NaïveRA for top-1 and top-10 queries, respectively. With SEIL, the improvement is by 18.8% and 8.0%, respectively.

Moreover, SEIL effectively reduces redundant distance computation. For top-1 and top-10 queries, SEIL reduces DCO by 4.1%–12.0% for NaïveRA, SRAIR, and RAIR.

Finally, comparing RAIR and SRAIR, we see that RAIR achieves lower DCO. When the first and the second assigned lists are the same, RAIR performs single assignment, while SRAIR strictly selects a second list from the rest of the lists to assign the vector. The results in Figure 14a indicates that this strict strategy is sub-optimal.

Memory Cost. Figure 14b displays the memory cost of NaïveRA, SRAIR, and RAIR with and without SEIL on SIFT1M. Table 4 reports the memory cost of the baseline IVPQfs and four redundant assignment solutions for five representative data sets. Since the refine layer is the same, we exclude the refine layer and report only the space of the IVF-PQ module.

We see that NaïveRA doubles the memory space used in the baseline IVPQfs due to redundant assignment. SEIL stores shared blocks of large cells only once, thereby significantly reducing the memory cost of redundant assignment. Overall, SEIL reduces the memory cost by 6.4%–42.5% for NaïveRA, SRAIR, and RAIR.

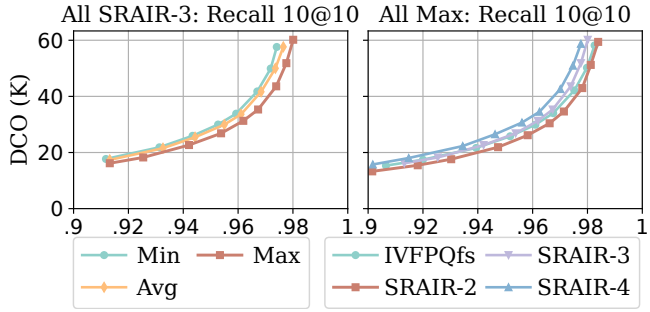
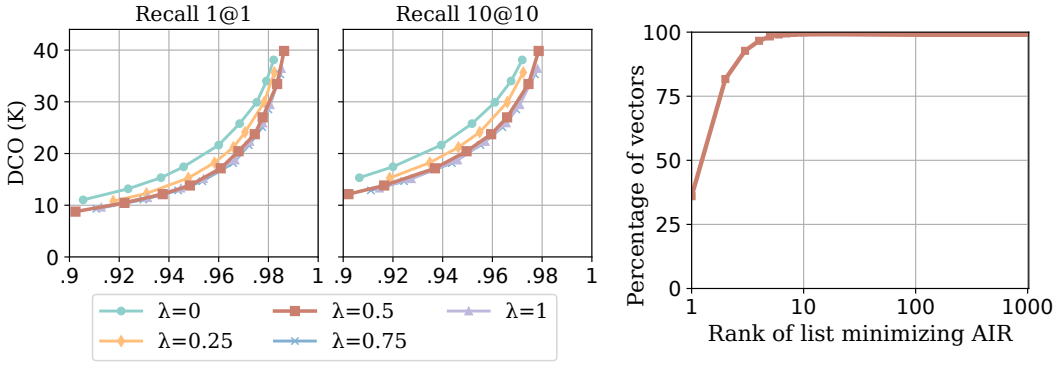


Fig. 15. Multiple assignment on SIFT1M.



(a) Recall-DCO curve varying λ (b) Distribution of rank of list minimizing AIR

Fig. 16. Parameter study on SIFT1M.

Note that the refine layer stores all the vector data. Compared to the refine layer, the baseline IVF-PQ module takes 6.4%–7.5% space, and the additional space for RAIRS is 1.2%–3.7%. Thus, RAIR trades off slight extra space for significantly better ANNS performance.

Multiple Assignment. The left part of Figure 15 compares the three aggregation functions (i.e., *max*, *min*, and *avg*) for 3-assignment on SIFT1M. *max* achieves the lowest DCO. The right part of Figure 15 compares single assignment (i.e., IVFPQfs), 2-assignment, 3-assignment, and 4-assignment. We choose the *max* aggregation function. SEIL is disabled since it is designed for 2-assignment. We consider the strict strategy (i.e., SRAIR) because RAIR essentially blurs the distinction among multiple assignments. From the figure, we see that SRAIR-2 is the best performing. This result indicates that over two assignment is unnecessary.

Parameter Study for λ. Figure 16a shows recall-DCO curves of RAIRS on SIFT1M while varying λ from 0 to 1. As λ increases, the curve shifts to the bottom-right, showing improved performance. The curve stops improving when λ reaches 0.5. Thus, we set the default value of λ to 0.5.

Parameter Study for N_CANDS. We modify Algorithm 3 so that *RairAssign* retrieves all the lists as the candidates in the ascending order of the distance between the list centroid and the vector to insert, and computes the list that minimizes the AIR metric. This gives the true rank in the sorted lists that should be assigned by AIR. Figure 16b depicts the Cumulative Distribution Function (CDF) of the true rank for all vectors in SIFT1M. We see that in 99.95% of the cases, the true rank ≤ 10. This means that considering the top-10 nearest lists achieves the correct assignment for most cases. Hence, we set $N\_CANDS = 10$ by default.

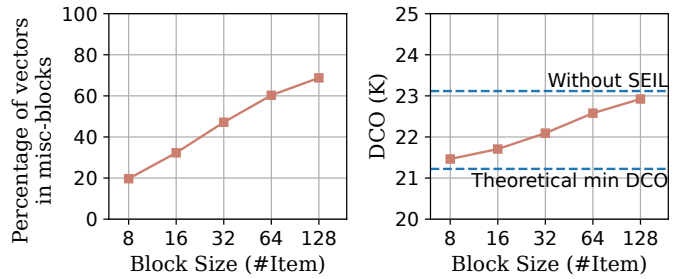


Fig. 17. Varying block size. (Default block size = 32)

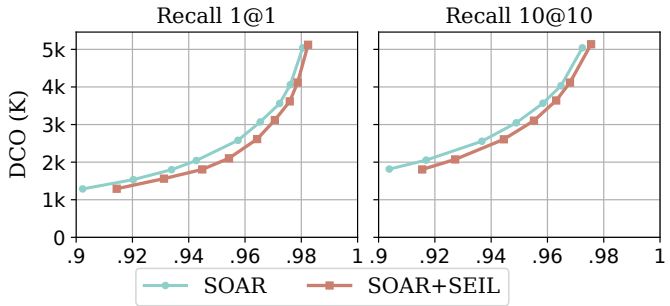


Fig. 18. Applying SEIL to SOAR on T2I.

Parameter Study for PQ Block Size. To understand the impact of block size on SEIL, we vary the block size, study the change of the number of vectors in misc-blocks, and compute the resulting DCOs for a given *nprobe*. As shown in Figure 17, as the block size increases, there are more vectors in misc-blocks. This is because the number of large cells (whose size ≥ block size) decreases. SEIL performs more redundant DCOs for the vectors in misc-blocks.

6.4 Applying SEIL to SOAR

We apply SEIL to SOAR, which targets the inner product distance. Figure 18 compares the recall-DCO curves of SOAR and SOAR+SEIL on T2I, which employs the inner product distance.

As shown in Figure 18, we see that SEIL significantly reduces the DCO of SOAR. This result indicates the applicability of the SEIL list layout optimization to various redundant assignment strategies and distance metrics.

7 Related Work

ANNS Methods. ANNS support has been implemented in both vector databases and general-purpose database systems [25, 34, 52, 54, 59, 63, 68]. Existing ANNS methods are divided into four categories: tree-based [10, 33, 41, 42, 65], hashing-based [2, 9, 35, 38, 53, 56, 60], graph-based [20, 28, 39, 44, 45, 49, 51, 69], and cluster-based [7, 26, 27, 47] approaches. IVF-based ANNS methods [47], which are a representative type of cluster-based approaches, are among the most widely used and highest performing ANNS methods. In this paper, we mainly focus on IVF-based ANNS.

Optimization Approaches for IVF-Based ANNS Methods. There are four main approaches to optimizing IVF-based ANNS: 1) quantization, 2) distance computation optimization, 3) hardware acceleration, and 4) redundant assignment. We review related work on approach 1–3 in the following. Redundant assignment is the focus of this work. We have discussed and experimentally compared the existing redundant assignment strategies.

Approach 1: Quantization. Quantization techniques are widely used to mitigate storage overhead and accelerate similarity search. Existing quantization methods can be categorized by the granularity

of dimensional grouping: single-dimension quantization (including VA_File [58], ITQ [24], LVQ [1], and RabitQ [22]), global quantization (including AQ [6] and LSQ [40]), and product quantization (including PQ [31], OPQ [23], PolysemousCodes [19], NEQ [15], DeltaPQ [55], VAQ [43], QAQ [67], and SparCode [48]).

Approach 2: Sampling-Based Distance Computation. Given the high dimensionality, distance computation often dominates the ANNS query response time [21]. ADSampling [21] leverages the Johnson–Lindenstrauss (JL) lemma and enable early termination by sampling a subset of dimensions during distance comparisons, effectively reducing the number of computed dimensions.

Approach 3: Hardware Acceleration. Various hardware features have been studied in the context of IVF-based ANNS. SPANN [13, 62] exploits SSDs to reduce the in-memory footprint and provide data persistence. GPUs [32] and FPGAs [16, 66] have been shown to significantly accelerate the ANNS processing. PQ Fast Scan [3], Bolt [11], and Quicker ADC [4] exploit SIMD instructions to accelerate the distance computation.

Relationship to Our Work: In general, our proposed ideas (i.e., RAIR and SEIL) are complementary to the above three optimization approaches. In this paper, we specifically use IVF with PQ quantization, refinement, and PQ fast scan as the baseline, which is one of the best performing IVF-based methods in practice. We have tailored our SEIL optimization to support PQ fast scan.

8 Conclusion

In conclusion, we propose RAIRS, combining two optimization techniques (i.e., RAIR and SEIL), for IVF-based ANNS in this paper. RAIR exploits redundant assignment to improve performance for queries that are poorly served by the first assigned lists. We formally derive the AIR list selection metric. Moreover, SEIL optimizes the list layout and exploits shared cells to reduce redundant distance computation. Our experimental results on representative real-world data sets confirm that RAIR and SEIL can effectively reduce DCO and improve ANNS performance for IVF-based ANN search.

Acknowledgments

This work is partially supported by National Key R&D Program of China (2023YFB4503600), Natural Science Foundation of China (62172390), and a collaboration project with Huawei Device Co., Ltd. Shimin Chen is the corresponding author.

References

[1] Cecilia Aguerrebere, Ishwar Singh Bhati, Mark Hildebrand, Mariano Tepper, and Theodore Willke. 2023. Similarity Search in the Blink of an Eye with Compressed Indices. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3433–3446.

[2] Alexandr Andoni and Piotr Indyk. 2008. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM* 51, 1 (2008), 117–122.

[3] 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. *Proceedings of the VLDB Endowment* 9, 4 (2015), 288–299.

[4] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2019. Quicker adc: Unlocking the hidden potential of product quantization with simd. *IEEE transactions on pattern analysis and machine intelligence* 43, 5 (2019), 1666–1677.

[5] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* 87 (2020), 101374.

[6] Artem Babenko and Victor Lempitsky. 2014. Additive quantization for extreme vector compression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 931–938.

[7] Artem Babenko and Victor Lempitsky. 2014. The inverted multi-index. *IEEE transactions on pattern analysis and machine intelligence* 37, 6 (2014), 1247–1260.

[8] Dmitry Baranchuk and Artem Babenko. 2021. *Text-to-Image dataset for billion-scale similarity search.* https://research.yandex.com/datasets/text-to-image-dataset-for-billion-scale-similarity-search

[9] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. 2005. LSH forest: self-tuning indexes for similarity search. In *Proceedings of the 14th international conference on World Wide Web*. 651–660.

[10] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.
[11] Davis W Blalock and John V Guttag. 2017. Bolt: Accelerated data mining with fast vector compression. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 727–735.
[12] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, et al. 2022. Improving language models by retrieving from trillions of tokens. In *International conference on machine learning*. PMLR, 2206–2240.
[13] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighborhood Search. *Advances in Neural Information Processing Systems* 34 (2021), 5199–5212.
[14] Thomas Cover and Peter Hart. 1967. Nearest neighbor pattern classification. *IEEE transactions on information theory* 13, 1 (1967), 21–27.
[15] Xinyan Dai, Xiao Yan, Kelvin KW Ng, Jiu Liu, and James Cheng. 2020. Norm-explicit quantization: Improving vector quantization for maximum inner product search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 51–58.
[16] Dimitrios Danopoulos, Christoforos Kachris, and Dimitrios Soudris. 2019. Fpga acceleration of approximate knn indexing on high-dimensional vectors. In *2019 14th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*. IEEE, 59–65.
[17] Matthijs Douze. 2024. *Guidelines to choose an index.* https://github.com/facebookresearch/faiss/wiki/Guidelines-to-choose-an-index
[18] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Herve Jégou. 2024. The Faiss library. (2024). arXiv:2401.08281 [cs.LG]
[19] Matthijs Douze, Hervé Jégou, and Florent Perronnin. 2016. Polysemous codes. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part II 14*. Springer, 785–801.
[20] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *Proceedings of the VLDB Endowment* 12, 5 (2019), 461–474.
[21] Jianyang Gao and Cheng Long. 2023. High-dimensional approximate nearest neighbor search: with reliable and efficient distance comparison operations. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.
[22] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.
[23] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized Product Quantization for Approximate Nearest Neighbor Search. In *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition*. 2946–2953.
[24] Yunchao Gong, Svetlana Lazebnik, Albert Gordo, and Florent Perronnin. 2012. Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval. *IEEE transactions on pattern analysis and machine intelligence* 35, 12 (2012), 2916–2929.
[25] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, et al. 2022. Manu: a cloud native vector database management system. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3548–3561.
[26] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*. PMLR, 3887–3896.
[27] Gaurav Gupta, Tharun Medini, Anshumali Shrivastava, and Alexander J Smola. 2022. Bliss: A billion scale index using iterative re-partitioning. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 486–495.
[28] Ben Harwood and Tom Drummond. 2016. Fanng: Fast approximate nearest neighbour graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 5713–5722.
[29] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 604–613.
[30] Herve Jegou and Laurent Amsaleg. 2010. *Datasets for approximate nearest neighbor search.* http://corpus-texmex.irisa.fr
[31] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2010), 117–128.
[32] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.
[33] Bastian Leibe, Krystian Mikolajczyk, and Bernt Schiele. 2006. Efficient clustering and matching for object class recognition.. In *BMVC*. 789–798.
[34] Jie Li, Haifeng Liu, Chuanghua Gui, Jianyu Chen, Zhenyuan Ni, Ning Wang, and Yuan Chen. 2018. The design and implementation of a real time visual search system on JD E-commerce platform. In *Proceedings of the 19th International*

Middleware Conference Industry. 9–16.
[35] Jinfeng Li, Xiao Yan, Jian Zhang, An Xu, James Cheng, Jie Liu, Kelvin KW Ng, and Ti-chung Cheng. 2018. A general and efficient querying method for learning to hash. In *Proceedings of the 2018 International Conference on Management of Data*. 1333–1347.
[36] Thomas Lidy. 2015. *Million Song Dataset Benchmarks*. https://www.ifs.tuwien.ac.at/mir/msd
[37] Ying Liu, Dengsheng Zhang, Guojun Lu, and Wei-Ying Ma. 2007. A survey of content-based image retrieval with high-level semantics. *Pattern recognition* 40, 1 (2007), 262–282.
[38] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases*. 950–961.
[39] Yu A Malkov and Dmitry A Yashunin. 2020. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2020), 824–836.
[40] Julieta Martinez, Joris Clement, Holger H Hoos, and James J Little. 2016. Revisiting additive quantization. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part II 14*. Springer, 137–153.
[41] Marius Muja and David G Lowe. 2009. Fast approximate nearest neighbors with automatic algorithm configuration.. In *International Conference on Computer Vision Theory and Applications*, Vol. 2. 2.
[42] Marius Muja and David G Lowe. 2014. Scalable nearest neighbor algorithms for high dimensional data. *IEEE transactions on pattern analysis and machine intelligence* 36, 11 (2014), 2227–2240.
[43] John Paparrizos, Ikraduya Edian, Chunwei Liu, Aaron J Elmore, and Michael J Franklin. 2022. Fast adaptive similarity search through variance-aware quantization. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2969–2983.
[44] Yun Peng, Byron Choi, Tsz Nam Chan, Jianye Yang, and Jianliang Xu. 2023. Efficient approximate nearest neighbor search in multi-dimensional databases. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.
[45] Jie Ren, Minjia Zhang, and Dong Li. 2020. HM-ANN: efficient billion-point nearest neighbor search on heterogeneous memory. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, Vol. 33. 10672–10684.
[46] J Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. 2007. Collaborative filtering recommender systems. In *The adaptive web: methods and strategies of web personalization*. Springer, 291–324.
[47] Sivic and Zisserman. 2003. Video Google: A text retrieval approach to object matching in videos. In *Proceedings ninth IEEE international conference on computer vision*. IEEE, 1470–1477.
[48] Liangcai Su, Fan Yan, Jieming Zhu, Xi Xiao, Haoyi Duan, Zhou Zhao, Zhenhua Dong, and Ruiming Tang. 2023. Beyond Two-Tower Matching: Learning Sparse Retrievable Cross-Interactions for Recommendation. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 548–557.
[49] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishaswamy, and Harsha Vardhan Simhadri. 2019. DiskANN: fast accurate billion-point nearest neighbor search on a single node. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. 13766–13776.
[50] Philip Sun, David Simcha, Dave Dopson, Ruiqi Guo, and Sanjiv Kumar. 2023. SOAR: improved indexing for approximate nearest neighbor search. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*. 3189–3204.
[51] Godfried T Toussaint. 1980. The relative neighbourhood graph of a finite planar set. *Pattern recognition* 12, 4 (1980), 261–268.
[52] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*. 2614–2627.
[53] Jingdong Wang, Ting Zhang, Nicu Sebe, Heng Tao Shen, et al. 2017. A survey on learning to hash. *IEEE transactions on pattern analysis and machine intelligence* 40, 4 (2017), 769–790.
[54] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. 2024. Starling: An I/O-Efficient Disk-Resident Graph Index Framework for High-Dimensional Vector Similarity Search on Data Segment. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–27.
[55] Runhui Wang and Dong Deng. 2020. DeltaPQ: lossless product quantization code compression for high dimensional similarity search. *Proceedings of the VLDB Endowment* 13, 13 (2020), 3603–3616.
[56] Xin-Jing Wang, Lei Zhang, Feng Jing, and Wei-Ying Ma. 2006. Annosearch: Image auto-annotation by search. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, Vol. 2. IEEE, 1483–1490.
[57] Roger Weber, Hans-Jörg Schek, and Stephen Blott. 1998. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, Vol. 98. 194–205.

[58] Roger Weber, Hans-Jörg Schek, and Stephen Blott. 1998. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, Vol. 98. 194–205.
[59] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. AnalyticDB-V: a hybrid analytical engine towards query fusion for structured and unstructured data. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3152–3165.
[60] Yair Weiss, Antonio Torralba, and Rob Fergus. 2008. Spectral hashing. *Advances in neural information processing systems* 21 (2008).
[61] Yuhuai Wu, Markus N Rabe, DeLesley Hutchins, and Christian Szegedy. 2022. Memorizing transformers. *arXiv preprint arXiv:2203.08913* (2022).
[62] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, et al. 2023. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 545–561.
[63] Wen Yang, Tao Li, Gai Fang, and Hong Wei. 2020. Pase: Postgresql ultra-high-dimensional approximate nearest neighbor search extension. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 2241–2253.
[64] Zehai Yang and Shimin Chen. 2026. RAIRS: Optimizing Redundant Assignment and List Layout for IVF-Based ANN Search. (2026). arXiv:2601.07183 [cs.DB]
[65] Peter N Yianilos. 1993. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the fourth annual ACM-SIAM symposium on Discrete algorithms*. 311–321.
[66] Jialiang Zhang, Jing Li, and Soroosh Khoram. 2018. Efficient Large-Scale Approximate Nearest Neighbor Search on OpenCL FPGA. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE, 4924–4932.
[67] Jin Zhang, Defu Lian, Haodi Zhang, Baoyun Wang, and Enhong Chen. 2023. Query-aware quantization for maximum inner product search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 4875–4883.
[68] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, et al. 2023. {VBASE}: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 377–395.
[69] Weijie Zhao, Shulong Tan, and Ping Li. 2020. Song: Approximate nearest neighbor search on gpu. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1033–1044.
[70] Zilliztech. 2023. *VectorDBBench: A Benchmark Tool for VectorDB.* https://github.com/zilliztech/VectorDBBench

Received July 2025; revised October 2025; accepted November 2025