

AKER: Density-Aware Approximate Caching for Vector Search

Sukjoon Oh*

KAIST

sjoon@kaist.ac.kr

Minki Kang

KAIST

minki.kang@kaist.ac.kr

Dohyun Kim

KAIST

ehgus421210@kaist.ac.kr

Baotong Lu[†]

Microsoft Research

baotonglu@microsoft.com

Jing Liu

Microsoft Research

jingliu3@microsoft.com

Qianxi Zhang

Microsoft Research

qiazh@microsoft.com

Qi Chen

Microsoft Research

cheqi@microsoft.com

Youjip Won[†]

KAIST

ywon@kaist.ac.kr

ABSTRACT

Disk-based approximate nearest neighbor search (ANNS) incurs high I/O overhead due to frequent disk accesses during index traversal. Approximate caching, which reuses the results of past queries to serve future similar queries, offers a promising approach to bypass disk searches. However, existing approaches suffer from two limitations. First, their hit predicates fail to simultaneously achieve high throughput and high accuracy, as they do not adapt to the varying local neighbor density. Second, they lack an effective refresh mechanism to maintain cache correctness during vector updates.

We present AKER, an approximate cache for disk-based ANNS. AKER addresses these limitations through two core design choices. First, we introduce a per-query similarity threshold that each cache entry dynamically adjusts based on cache hit patterns. This design enables AKER to adapt to neighborhood densities to preserve both efficiency and accuracy. Second, we propose *del-consistency*, a consistency model that applies deletions eagerly and insertions lazily. Under this model, AKER implements a low-overhead refresh mechanism that bounds cache staleness while preserving search accuracy. We integrate AKER into pgvector and evaluate it on representative workloads. AKER improves recall by up to 64 percentage points over prior solutions and increases QPS by up to 3.2 $\times$, while using 0.6 $\times$ the memory of pgvector’s shared buffers.

PVLDB Reference Format:

Sukjoon Oh, Minki Kang, Dohyun Kim, Baotong Lu, Jing Liu, Qianxi Zhang, Qi Chen, and Youjip Won. AKER: Density-Aware Approximate Caching for Vector Search. PVLDB, 19(10): 2727 - 2740, 2026.
doi:10.14778/3828612.3828627

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/sjoon-oh/aker>.

1 INTRODUCTION

Vector search has become a foundational component in modern data-intensive applications, spanning web search engines [11, 22],

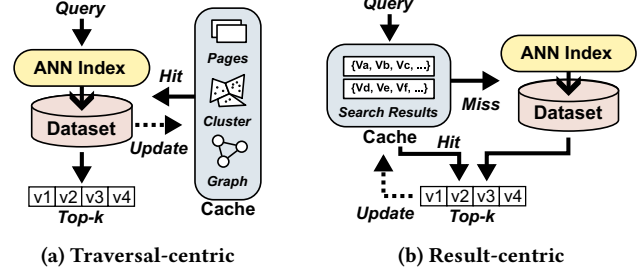


Figure 1: Two categories of caching strategies for ANNS.

recommendation systems [25, 37, 39, 41, 58], scientific discovery [16], and retrieval-augmented generation (RAG) [14, 18, 38]. These workloads rely on approximate nearest neighbor search (ANNS) to efficiently probe million- to billion-scale vector collections and retrieve relevant results under sub-second latency constraints. At the core of ANNS is the *vector index*, which organizes high-dimensional vectors according to their proximity, enabling efficient query processing while preserving high search accuracy.

As data volumes continue to grow, modern vector stores place both vectors and indexes on disk at low cost [1, 3, 9, 21, 22, 32, 56, 60]. However, disk-resident storage makes I/O the dominant performance bottleneck during vector search. We observe that this bottleneck primarily stems from the inherent *read amplification* in ANNS: to satisfy accuracy requirements, a query typically accesses orders of magnitude more data than is ultimately returned. Specifically, the query first traverses a disk-resident index – organized as clusters [22, 26, 60] or graphs [32, 44] – during which candidates are fetched from disk into memory for distance computation, and then re-ranks the visited candidates to produce the top-k. Achieving higher accuracy targets requires more aggressively expanding the search scope during index traversal, which substantially increases disk I/O and further exacerbates read amplification [63].

To mitigate this bottleneck, prior work has improved ANNS performance through algorithmic optimizations [22, 32, 60, 63] or emerging hardware [31, 46, 53, 54]. Complementary to these efforts, this paper investigates a practical result-centric cache for disk-based ANNS, with the goal of reusing past query results in DRAM to improve end-to-end query performance.

Figure 1 summarizes two caching strategies for ANNS: *traversal-centric* caching and *result-centric* caching. Traversal-centric caches [9, 32, 46, 62] retain intermediate data accessed during index traversal in memory, such as pages, clusters, or graph neighbors. By reusing

*Work partially performed during the internship at Microsoft Research.

[†]Corresponding authors: Baotong Lu and Youjip Won.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 19, No. 10 ISSN 2150-8097.

doi:10.14778/3828612.3828627

the cached index structures and vectors, this approach reduces disk I/O during query processing. However, sustaining high performance typically requires a large cache budget, as ANNS traversal often incurs a high memory footprint due to excessive vector accesses.

In contrast, result-centric caches [23, 27, 45, 47, 49] store only the past query vectors and their top- k results. To serve an incoming query, the system evaluates its similarity to cached queries and returns the corresponding top- k results when the similarity exceeds a predefined threshold. As this strategy does not require strict query equivalence, it constitutes an *approximate cache* relying on semantic similarity between queries. By avoiding the storage of intermediate traversal data, result-centric caching significantly reduces memory consumption. Moreover, cache hits can entirely bypass expensive ANN index traversals, yielding substantial performance gains. As a result, result-centric caching represents a promising direction for improving ANNS efficiency. However, the broader hit predicate makes cache hits no longer accuracy-safe, introducing two challenges that limit existing designs in practice.

Dilemma between cache efficiency and accuracy. Since result-centric caching relies on query similarity, it faces a trade-off between cache efficiency and accuracy. A loose similarity threshold increases cache hit ratios and thus improves system efficiency, but may match insufficiently similar queries, degrading accuracy. Conversely, a tight threshold preserves search accuracy but may lead to low cache hit ratios by missing potential reuse opportunities. Furthermore, vector distributions in the high-dimensional space are often highly skewed: some regions are dense with many vectors, while others are sparse. This heterogeneity renders the one-size-fits-all similarity thresholds adopted by prior work [18, 23, 27, 30, 45, 49] ineffective in balancing efficiency and accuracy across varying vector densities in the vector space.

Heavyweight cache refresh. Modern vector databases continuously receive update operations [51, 60], requiring cached results to be refreshed to avoid staleness and maintain correctness. Otherwise, caches may miss newly inserted vectors or return deleted items. However, supporting cache refresh is expensive: it either requires scanning all cached entries to determine whether their results should be updated or aggressively invalidating cached entries, which forces costly ANNS executions in the underlying database. Consequently, existing result-centric caching approaches [18, 30] choose to permit stale entries at the expense of ANN accuracy or degrade cache efficiency under frequent updates.

This paper presents AKER, an approximate cache that makes similarity-based reuse practical for disk-based vector searches by balancing accuracy and efficiency while supporting lightweight cache refresh. AKER is built on two core ideas. First, AKER introduces a *density-aware caching strategy* which maintains an individual similarity threshold for each cached query. Specifically, AKER adapts the threshold based on the local data distribution: it tightens the threshold for queries located in dense regions to preserve accuracy, while relaxing it in sparse regions to improve cache utilization. Moreover, the similarity threshold is dynamically adjusted at runtime, with accuracy treated as a first-class optimization objective. Second, AKER introduces *del-consistency*, a practical consistency model between the cache and the underlying vector database. Del-consistency enforces immediate invalidation of deleted items to

prevent stale results from being returned, while allowing the visibility of insertions to be deferred for update efficiency, which aligns well with the approximate nature of ANNS.

We implement AKER through carefully designed caching structures that achieve high memory efficiency and fast cache-side searches. To reduce metadata overhead for queries sharing identical top- k results, AKER organizes cached queries into representative entries and lightweight alias entries. Only representative entries store the top- k results, while alias entries are linked to their corresponding representatives. To support efficient lookup for similar queries, AKER maintains a query filter which builds an embedded ANN index over cached queries, enabling the retrieval of similar queries without brute-force scans. Finally, AKER implements del-consistency via a lightweight refresh pipeline. We execute deletions in the cache by removing deleted vectors from cached results and repairing affected top- k lists using a small pool of top- $(k + \Delta)$ candidates, avoiding costly ANNS traversal in the underlying database. In contrast, insertion visibility is safely deferred by tracking an accuracy degradation risk for each cached query, triggering top- k recomputation only when the estimated risk exceeds a threshold.

We make three contributions. First, we identify two fundamental limitations of existing approximate caches for ANNS and highlight the design challenges. Second, we propose AKER, an approximate caching system that combines density-aware similarity thresholds to balance accuracy and efficiency with del-consistency to efficiently support cache refresh under continuous updates. Finally, we implement AKER with memory-efficient caching structures that support fast lookups and lightweight refresh pipelines.

By integrating AKER into the popular PostgreSQL pgvector extension [1–3, 9], we verify the effectiveness of AKER through end-to-end evaluations. Across both web search and RAG workloads, AKER improves accuracy by up to 64 percentage points over prior state-of-the-art approximate caches. Moreover, compared to pgvector’s default shared buffer (i.e., traversal-centric caching), AKER reduces read amplification by up to 0.4 $\times$ and increases throughput by up to 3.2 $\times$ while using only 0.6 $\times$ of the memory.

2 BACKGROUND

2.1 Vector Search and Index

Vector search is widely used in modern data-intensive applications such as web search [11, 22] and retrieval-augmented generation [14, 18, 38]. In these systems, each data item is represented as a vector in a high-dimensional space. Given a query vector q , the system retrieves the top- k most similar vectors according to a similarity metric¹ such as Euclidean, cosine, or inner product [9, 18, 22, 26].

Finding nearest neighbors via a brute-force scan yields exact results but incurs prohibitively high computational cost, especially at large scale. Consequently, modern vector stores adopt approximate nearest neighbor search (ANNS) to trade result accuracy for high search efficiency [9, 18, 22, 26, 32, 60]. ANNS systems rely on vector indexes that organize vectors based on proximity for efficient retrieval of target vectors. Existing ANN indexes can be broadly categorized into two classes: cluster-based [9, 22, 26, 60] and graph-based [32, 44, 46, 51, 53, 56]. Cluster-based indexes group similar

¹We use similarity and distance interchangeably in this paper. Higher similarity between vectors indicates smaller distance in the vector space.

vectors using clustering algorithms such as k -means [22, 26, 60], with each cluster represented by a centroid. At query time, the system probes a small subset of clusters whose centroids are closest to the query vector. In contrast, graph-based indexes (e.g., HNSW [44]) construct a proximity graph in which each vector is connected to its nearest neighbors. Queries are processed via greedy graph traversal starting from one or more entry points, progressively moving toward closer vectors while pruning unrelated regions of the graph.

The accuracy of ANNS is commonly evaluated using $\text{recall}@k = \frac{|R_a \cap R_e|}{k}$, where R_a denotes the set of k searched vectors and R_e denotes the ground-truth top- k results. In practice, ANN indexes expose a trade-off between search accuracy and efficiency metrics such as throughput (QPS) and latency [22, 26, 32, 44, 60, 63]. Owing to their cost-effectiveness and competitive performance, disk-based ANNS solutions are widely adopted in vector databases [4, 7, 9, 10, 55] and have been deployed in major cloud services [1, 3].

2.2 Caching Strategy for Vector Search

To reduce I/O costs, disk-based ANNS systems employ DRAM to cache frequently accessed data [9, 22, 32, 46, 56, 60]. As illustrated in Figure 1, existing caching mechanisms for ANNS can be broadly categorized into two types – *traversal-centric* caching and *result-centric* caching – based on the type of data being cached.

Traversal-centric caches store index structures and vectors accessed during ANNS traversal. For instance, in cluster-based ANN indexes, the cache typically contains centroids and accessed clusters, whereas in graph-based indexes it stores frequently visited neighbors. Some approaches adopt static caching strategies [32], which place vectors near the entry points directly in DRAM, without cache replacement between disk and memory. Other systems, such as PostgreSQL pgvector [9], leverage the traditional database buffer pool to dynamically cache the hot data. Traversal-centric caching is effective when the dataset or index structures are small enough to fit in memory, or when sufficient DRAM is available to cache a substantial fraction of the traversal-related data.

In contrast, result-centric caches aim to reduce memory consumption by storing only previously issued query vectors along with their final top- k results. When processing an incoming query, the system computes its similarity to cached queries and directly returns the corresponding top- k results if the queries match exactly or if their similarity meets a predefined threshold. Since the value of k is typically fixed by the application and queries are often skewed, result-centric caching has proven effective in practice and been adopted in various applications [19, 29, 30].

2.3 Approximate Cache

Beyond serving equal-match queries, result-centric caches for ANNS provide a unique opportunity to reuse search results across queries that are semantically similar [52]. This opportunity arises from a fundamental property of modern embedding models: vectors that are close in the embedding space typically correspond to semantically similar texts or images. Consequently, even when two query vectors are not identical but sufficiently close, their top- k nearest neighbors in the database often exhibit high overlap.

This observation enables result-centric caches to operate as an *approximate cache*, in which the result returned from the cache may

differ from that produced by the underlying ANNS engine. In such a cache, query similarity serves as the cache-hit predicate. Let q denote a cached query and A its associated top- k results. When a new query q' arrives, the cache returns A if the distance between q and q' , denoted by $d(q, q')$, falls below a predefined threshold τ . The choice of τ is therefore critical, as it directly governs the trade-off between result accuracy and cache efficiency (hit ratios).

We present two representative approximate cache designs: PROXIMITY [18] and Potluck [30]. PROXIMITY [18] is an approximate cache designed for RAG systems, where each cache entry stores a query vector along with its top- k search results. Cache hits are determined using a fixed similarity threshold τ , which is configured at system initialization and remains unchanged during runtime. While this design is simple and easy to deploy, the static threshold makes it difficult to balance accuracy and efficiency under varying query workloads and data distributions.

Potluck [30] is an approximate cache originally proposed to share computation results across mobile applications, but it is general enough to apply to ANNS workloads. Unlike PROXIMITY, Potluck dynamically adjusts the similarity threshold during runtime to improve cache effectiveness. To this end, Potluck introduces a *dropout* mechanism to estimate the accuracy of cached results. When a query hits the cache, Potluck probabilistically triggers an intentional cache miss and executes the query against the underlying ANNS engine. The returned result is then compared with the cached result. If the two results match, indicating high cache accuracy, τ is increased to improve the cache hit ratio for future queries; otherwise, it is decreased to reduce approximation error.

3 MOTIVATION AND CHALLENGES

3.1 Read Amplification in ANNS

We observe that the I/O bottleneck in disk-based ANNS primarily stems from excessive vector accesses during index traversal, particularly when high recall is required. To quantify this effect, we define read amplification ρ as the ratio between the total size of vectors accessed during index traversal, denoted by C , and the size of the top- k results read from disk, denoted by R : $\rho \triangleq \frac{C}{R}$. For traditional indexes over scalar data types, such as LSM-trees, read amplification is effectively bounded on the order of a few tens (e.g., 20–35 $\times$) due to simplified traversal paths [20, 36, 42, 48, 64]. In contrast, disk-resident ANNS exhibits substantially higher read amplification, as vector queries often need to access a large number of neighboring vectors to avoid missing potential candidates.

To study this behavior, we measure the read amplification of two representative disk-based ANNS systems – PostgreSQL pgvector [9] and SPANN [22] – using the BIGANN [5] and SPACEV [11] datasets. We cover two vector indexes supported in pgvector: the graph-based HNSW and the cluster-based IVF. During query processing, pgvector relies on PostgreSQL’s shared buffer pool to load index structures and vector data from disk and cache frequently accessed pages. Therefore, the shared buffer pool effectively functions as a traversal-centric cache. SPANN [22] is a cluster-based system that adopts a static caching strategy, where cluster centroids are retained in memory while all posting lists reside on disk.

Figure 2 reports the read amplification of different indexes under varying recall requirements. Even at a low recall of 0.5, pgvector’s

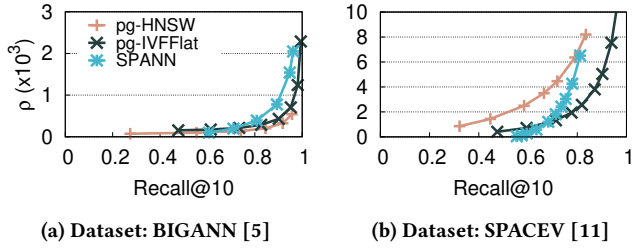


Figure 2: Read amplification (ρ) of disk-based ANNS systems under different recalls. The amplification becomes extremely severe at high recall requirements (e.g., 0.8–0.9).

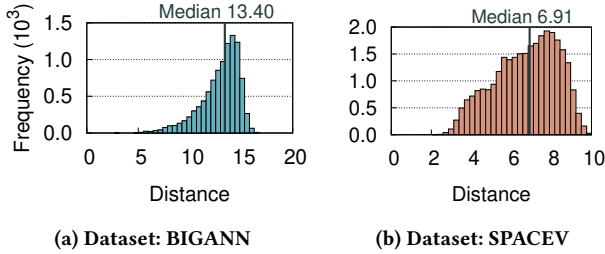


Figure 3: Distribution of ground-truth top-1 distances to the queries. Large distance implies even the nearest neighbor is far away; thus the query lies in a sparser region.

HNSW (pg-HNSW) and IVFFlat (pg-IVFFlat) already exhibit significant read amplification, reaching $70\times$ and $155\times$ across both datasets. Read amplification worsens as the recall requirement increases. To improve recall from 0.5 to 0.8, pg-HNSW incurs at least $7.4\times$ and $9.5\times$ higher disk I/Os on the two datasets, while pg-IVFFlat requires $14.7\times$ and $24.6\times$ more, respectively. SPANN shows a comparable trend, suffering from similarly severe read amplification. Therefore, meeting application-level recall targets (e.g., 0.8–0.9) can result in read amplification on the order of 10^2 – $10^3\times$.

The root cause is that achieving high recall necessitates accessing a much larger number of candidate vectors to avoid missing true nearest neighbors. For instance, on the SPACEV dataset, to return top-10 results occupying 4 KB, pgvector reads an average of 29.7 MB of data per query to achieve $recall = 0.9$. This mismatch highlights the inefficiency of disk accesses at high recall. While in-memory ANNS solutions eliminate disk I/O by keeping all data in DRAM, they scale poorly to large datasets due to prohibitive memory costs [26, 32, 46]. As an alternative, caching query results in memory offers an attractive design point: upon a cache hit, the system can bypass index traversal entirely, thereby substantially reducing read amplification and end-to-end query latency.

3.2 A Single Global Threshold for All Queries

Prior approximate caches rely on a single similarity threshold to determine whether an incoming query can become a cache hit. The similarity threshold τ is either globally fixed, or periodically adjusted. However, the single global τ struggles to simultaneously achieve high cache efficiency and accuracy. This limitation stems from the inherent non-uniformity of vector spaces: the local vector

density – the number of vectors within a given radius r – can vary substantially across different regions [27]. A global threshold implicitly assumes that a single reuse radius is suitable for all queries and enforces a trade-off. Specifically, a larger τ improves cache hit ratios by treating more queries as hits, but risks returning incorrect neighbors in dense regions; conversely, a smaller τ preserves accuracy while causing unnecessary cache misses in sparse regions.

To quantify this effect, we empirically examine density variance using the BIGANN and SPACEV datasets.² Because selecting a meaningful radius r is non-trivial, we approximate local density using the distance from each query to its closest ground-truth neighbor. For a query q , this distance is the radius of the smallest hypersphere centered at q that contains at least one data point. Accordingly, smaller distances indicate denser regions of the vector space, while larger distances correspond to sparser regions.

As shown in Figure 3, the top-1 distances range from 2.7 to 16.7 on BIGANN and from 4.2 to 10.0 on SPACEV, indicating substantial variation in local density. A single threshold thus cannot adapt to diverse query characteristics. In Section 5.2, we empirically verify that no single threshold can simultaneously balance recall and cache hit rate. This observation motivates our design of a *density-aware similarity threshold*, which maintains per-query thresholds and hit conditions and dynamically adjusts each threshold at runtime.

3.3 Absence of Cache Refresh Scheme

When the underlying ANN indexes evolve due to insertions and deletions, the cache must reflect these changes to prevent staleness. Existing approximate caching schemes typically assume a static dataset and therefore lack explicit cache refresh mechanisms. However, real-world workloads may continuously process insert and delete operations to maintain dynamic vector collections [40, 57, 60]. In those workloads, if a vector closer to a query q is newly inserted, its cached top- k results must be updated to preserve search quality. Similarly, when one or more vectors in the cached top- k are deleted, the cache must be refreshed to reflect the changes.

Refreshing cached results in approximate caches is non-trivial. Consider a query q whose cached top-3 results are v_a, v_b, v_c with distances 0.13, 0.24, and 0.31, respectively. If a new vector v_d is inserted with distance 0.19 to q , the correct top-3 becomes v_a, v_d, v_b . Updating the cache accordingly, however, can incur substantial overhead. Specifically, the system must first determine which cached queries are affected by the insertion, which requires scanning all cached queries and computing their similarity to the newly inserted vector. Deletions introduce analogous challenges: the system must identify cache entries whose top- k results include deleted vectors. Moreover, deletions invalidate cached queries because they reduce the number of available results below k . This requires the system to re-execute ANNS in the underlying database to reconstruct the top- k .

To efficiently maintain cache consistency under dynamic updates, we propose a result-level consistency model, termed *del-consistency*. Based on this model, we further design a lightweight cache refresh mechanism that incurs low maintenance overhead while effectively bounding recall degradation.

²For simplicity and clarity of presentation, we use Euclidean distance throughout the paper, where smaller values indicate higher similarity between vectors. Our design can be naturally extended to other distance metrics, such as cosine distance.

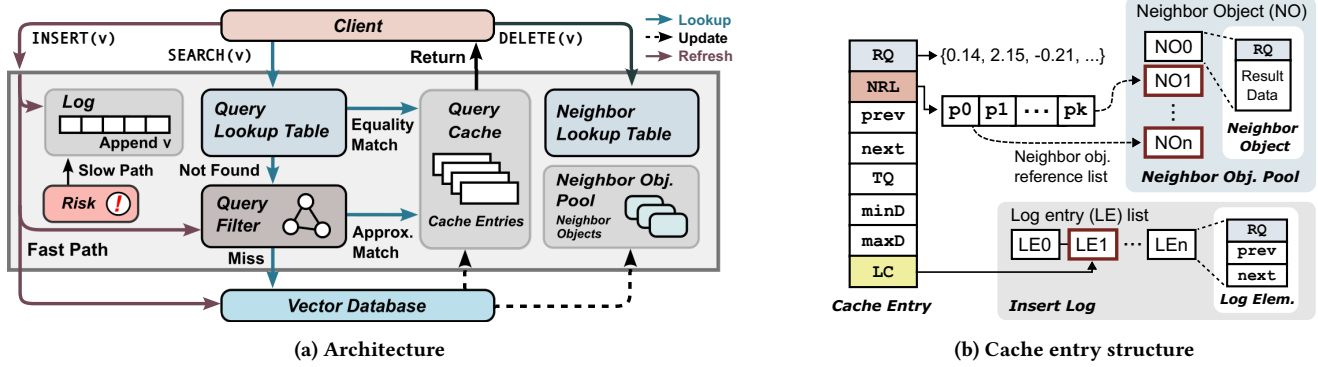


Figure 4: AKER overview.

4 AKER

4.1 Overview

We present AKER, an approximate result cache for disk-based ANNS systems. AKER combines density-aware reuse control with light-weight refresh to support similarity-based result caching under dynamic updates. Figure 4a illustrates key components of AKER. For searches, AKER probes the cache and returns a cache entry on a hit; otherwise, it queries the vector database. For inserts and deletes, AKER refreshes cached results to reflect updates in the base data.

Vector search. AKER handles search requests with equality hits and approximate hits. It first searches the query lookup table, a hash table that maps a hashed query vector to a cache entry. Each cache entry stores the query vector and its top- k results. AKER first searches for an identical query in the query lookup table. When a match is found, the cache entry is returned. Otherwise, AKER takes an approximate search path by querying a *query filter*.

The query filter is a small ANN index built over the query vectors of the cache entries. Its size depends on the number of cache entries. The query filter serves two roles: (1) retrieving a cache entry whose query is similar to an incoming query without a brute-force scan, and (2) locating cache entries that are likely to be affected by update requests. The cache entry is returned if an approximate hit condition is met; otherwise, the request becomes a miss.

Vector insertion. When a client inserts a new vector into the vector database, AKER records the vector in an *insert log*. AKER then refreshes affected cache entries via two paths: a *fast path* and a *slow path*. The fast path uses the query filter to locate the cache entry whose query is closest to the inserted vector. It checks whether the new vector would enter that cache entry’s top- k results; if so, AKER updates the cached top- k accordingly. The slow path processes insert log records lazily in batches. AKER estimates the risk of recall degradation from lazy refreshes. When the risk exceeds a predefined threshold, it refreshes unchecked log entries. The two paths refresh the cache without a full scan of all cache entries.

Vector deletion. Upon a vector deletion request, AKER first searches a neighbor lookup table to check whether the vector is present in the cache. Each neighbor object represents a cached result vector. If any of the vectors requested to be deleted reside in the cache, AKER immediately removes them and updates the neighbor lookup

Table 1: Summary of key designs in AKER.

	Description	Section
Representative/ alias cache entry	Representative entries hold results; alias entries point to the representative entries.	4.2, 4.4
Query filter	Embedded index of cache entries that finds similar cache entry candidates.	4.1, 4.4, 4.7
Threshold τ_q	Reuse radius for a cached query q ; enlarged and shrunk dynamically from hit types.	4.2, 4.3, 4.4
Refresh	Del-consistency combines immediate deletion with fast/slow refresh via risk tracking.	4.1, 4.6
Active/Standby filter	Two-filter scheme that separates cache entry insertions from evictions.	4.7

table. Cache entries that fail to keep a complete set of top- k are repaired during cache lookups by promoting reserved candidates.

The remainder of Section 4 explains AKER’s design in detail. Section 4.2 defines the cache entry structure, Section 4.3 presents the per-query similarity threshold mechanism, Sections 4.4, 4.5, and 4.6 describe cache lookups, updates, and refreshes, and Section 4.7 explains eviction and query filter maintenance.

4.2 Cache Entry Structure

Figure 4b shows the structure of a cache entry. The raw-query field RQ stores the query vector. The neighbor object reference list NRL stores references to the neighbor objects that store the query’s top- k results. The prev and next pointers link similar cache entries. The per-query similarity threshold τ_q is stored in the TQ field. The maxD and minD fields keep the maximum and minimum distance between the query and its cached top- k , respectively. AKER initializes TQ to minD/4, a conservative setting that biases early lookups toward preserving recall until AKER adapts TQ online.

Each element of NRL points to a neighbor object in a neighbor object pool. AKER allows multiple cache entries to share neighbor object references to save cache space. The cache size is controlled by capping the total number of neighbor objects. The neighbor lookup table is a hash table that maps each key vector to its corresponding neighbor object in the shared pool. Each neighbor object stores the raw vector and database-specific metadata (e.g., tuple identifiers

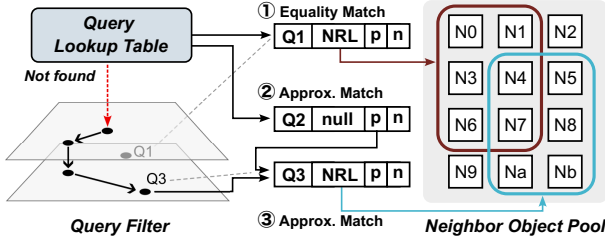


Figure 5: AKER lookup. AKER probes query lookup table: on equality match (1) returns the entry; on alias hit (2) follows prev to the representative. Otherwise (3), probes query filter and returns a nearest cache entry; otherwise it is a miss.

in PostgreSQL). Because a neighbor object can be referenced by multiple cache entries, AKER uses a reference counter for each neighbor object for safe memory reclamation. When inserting a new cache entry, AKER checks whether each top- k vector already exists in the neighbor lookup table; if so, it reuses the existing neighbor object and increments its reference counter. Otherwise, it creates a new neighbor object and inserts it into the table. When a vector deletion is requested, AKER removes the corresponding neighbor object from the table.

To avoid repeated approximate lookups for recurring queries, AKER uses *entry linking*. Specifically, when a query is served via an approximate hit, AKER registers a cache entry in the query lookup table. This newly inserted entry points to the cache entry that was previously served. When the same query vector arrives again, AKER directly returns the entry in $O(1)$ using the query lookup table, skipping both the query filter probe and the similarity check.

Cache entries are classified into two types in AKER: *representative* and *alias*. A representative entry owns the NRL and is indexed in the query filter. Each element in the representative entry NRL points to a neighbor object. In contrast, an alias entry keeps an empty NRL. It links to a representative entry and is not indexed in the query filter. Cache entries are linked via prev/next pointers, forming a list whose head is the representative entry. A cache lookup returns a representative entry either directly or by following prev from an alias entry to the list head. By indexing only representative entries in the query filter, AKER can keep the query filter small (Section 4.7), while allowing repeated queries that were previously served via approximate hits to be answered via the query lookup table.

The log checkpoint field LC references an insert log record. It tracks the progress of slow path refresh for a cache entry, i.e., how far the cache entry has processed insert log updates. When AKER invokes the slow path after vector insertions, it performs batched distance checks, scanning from LC (Section 4.6). Only representative entries maintain a valid LC, and alias entries set LC to null.

4.3 Per-Query Similarity Threshold

AKER maintains a per-query similarity threshold τ_q for each cached representative query q . The threshold τ_q controls the acceptance region for approximate reuse around q . When tuning τ_q , AKER prioritizes both low-overhead control and conservative safeguards. The most precise method is verification-based tuning. A cache can re-traverse the index and adjust the threshold based on the search discrepancy [30]. However, such index re-traversals on the hit path

Algorithm 1: Cache Lookup

Input: query q

```

1 Procedure cacheRead( $q$ ):
2    $e \leftarrow \text{QueryLookupTable}(q)$ 
3   if  $e == \text{NULL}$  then
4      $e \leftarrow \text{QueryFilter}(q)$ 
5     if  $\text{distance}(q, e.RQ) < e.TQ$  then
6        $e.TQ \leftarrow \text{updateTQ}(e.TQ)$ 
7     return  $e$ 
8   return NULL // miss
9   while  $e.\text{prev}$  is not NULL do
10     $e \leftarrow e.\text{prev}$  // Find representative entry
11     $e.TQ \leftarrow \text{updateTQ}(e.TQ)$ 
12  return  $e$ 

```

reduce the benefit of caching. Thus, AKER uses a low-overhead signal already present in the cache – the hit type – because hit types have different impacts on search quality.

An equality hit introduces no reuse error: the requested query is identical to the cached query, so returning the cached top- k reproduces the same ANNS result. Thus accuracy does not change in this case. In contrast, an approximate hit returns the cached result of another nearby query, so its ANNS results can differ and reduce accuracy. AKER therefore treats equality hits as a low-risk signal for cautious increase in reuse opportunities, while treating approximate hits as a risk signal for preserving accuracy. Concretely, AKER updates the per-query threshold TQ as follows:

$$\tau_{q,\text{next}} \leftarrow \begin{cases} \tau_{q,\text{prev}} \cdot \alpha_L, & (\alpha_L > 1), \text{ Equality-match} \\ \tau_{q,\text{prev}} \cdot \alpha_T, & (0 < \alpha_T < 1), \text{ Approximate-match} \end{cases} \quad (1)$$

Here, τ_q denotes the similarity threshold associated with a cached query q . $\alpha_L > 1$ and $\alpha_T \in (0, 1)$ are multiplicative factors for increasing and decreasing the threshold, respectively. To achieve better hit ratios, τ_q must be large. To achieve better search accuracy, τ_q must be kept small. AKER prioritizes search accuracy, thus using asymmetric update factors $\alpha_L = 1.1$ and $\alpha_T = 0.25$ as its default values. To prevent τ_q from growing far beyond a reasonable distance, AKER limits the maximum τ_q by the distance between q and its closest cached neighbor. This minimum distance acts as a per-query safety bound. Each τ_q is updated on every cache hit.

4.4 Cache Lookup

Figure 5 and Algorithm 1 present a cache lookup workflow. AKER distinguishes two cache-hit types: *equality match* and *approximate match*. An equality match happens when an incoming query is identical to the cached query vector. In the case of an equality match, AKER first probes the query lookup table (line 2). If an identical query vector is found, the corresponding cache entry is returned.

An approximate match occurs when the incoming query is not identical to the cached representative query but is sufficiently close. AKER supports two approximate match paths: (i) returning an alias entry via the query lookup table, or (ii) returning a representative entry via the query filter. First, the alias path handles queries that have previously produced an approximate hit. Specifically, suppose an incoming query q' approximately matches a cached query q

stored in a representative entry e . AKER then creates an alias entry e' for q' and registers it in the query lookup table. The alias entry e' stores a reverse pointer prev to its representative entry e . Upon a repeated request for the same vector q' , AKER finds e' from the query lookup table and follows prev to retrieve e (line 10), returning e 's NRL. This avoids rerunning the query filter search and distance comparison because q' is already similar to the previously validated representative entry.

If no alias entry is found in the query lookup table, AKER searches the query filter for a similar representative entry (line 4). Only representative entries are indexed in the query filter. Given a query q' , the query filter returns the nearest representative entry e whose cached query vector RQ is closest to q' . AKER then computes the distance between q' and RQ. If the distance is below the per-query similarity threshold TQ , it becomes an approximate hit.

4.5 Cache Update

To insert a new cache entry, AKER first generates neighbor objects for the retrieved top- k results and registers them in the neighbor lookup table. If an identical vector exists, AKER increments the reference counter of the existing neighbor object. AKER then creates a new representative entry and records the references in its NRL. The representative entry is registered in the query lookup table and indexed by the query filter. If the request is an approximate match, AKER creates an alias entry. AKER traverses the representative entry's next pointers and links the alias entry at the tail. Finally, AKER inserts the alias entry into the query lookup table.

AKER initializes the LC to the insert log tail. This is safe because the new representative entry is built over the latest database snapshot. In the slow refresh path, AKER scans the insert log starting from the cache entry's LC to determine whether the vector should be included in the top- k set. A representative entry created on a cache miss at time t is computed over the current database snapshot at time t . Thus, the search result reflects all latest inserts up to the insert log tail at time t .

4.6 Cache Refresh

Del-consistency. We define *consistency* for ANNS caches as the result-level visibility agreement between the vectors in the underlying database and the vectors in the cache at time t for a query q . Let V_t denote the set of live vectors in the database at time t , and let c_t denote the cached top- k results of q returned on a cache hit. Let s_t denote the results of q returned by executing the base ANNS on the database at time t . We propose a new consistency model called *del-consistency*. A cache is *del-consistent* at time t for any query q if it satisfies validity, while allowing relaxed visibility:

P 1 (VALIDITY). At any time t , cached results of q contain only valid vectors, i.e., $c_t \subseteq V_t$ and $|c_t| = k$.

P 2 (RELAXED VISIBILITY). At any time t , the cached results c_t are not required to equal the ANNS results s_t for the same query.

Validity requires that deletion operations be reflected immediately in NRL. Otherwise, the cache could return vectors that no longer exist in the database. *Relaxed visibility* allows temporary staleness with respect to newly inserted vectors, which aligns well

Algorithm 2: Cache Refresh (Insertions)

```

Input: new vector  $v$ 
1 Log  $\mathcal{L}$ ; Batch size  $B$ ; Risk  $R$ ; Risk threshold  $\tau_{\text{risk}}$ ;
2 Procedure fastPath( $v$ ):
3    $e \leftarrow \text{QueryFilter}(v)$  // fast path
4   if distance( $e, v$ ) <  $e.\text{maxD}$  then
5      $e.\text{NRL} \leftarrow \text{updateNRL}(v)$ 
6      $R \leftarrow \text{updateRisk}(e)$ 
7     if  $R \geq \tau_{\text{risk}}$  then
8       slowPath( $B$ )
9 Procedure slowPath( $B$ ):
10   $e_c \leftarrow \text{NextEntry}$  //  $e_c$ : Cache entry
11  for  $e_l = \mathcal{L}[e_c.LC]$  to  $\mathcal{L}[e_c.LC + B]$  do
12    if distance( $e_l.RQ, e_c.RQ$ ) <  $e_c.\text{maxD}$  then
13       $e_c.\text{NRL} \leftarrow \text{updateNRL}(e_l)$  //  $e_l$ : Log entry
14   $e_c.LC \leftarrow e_c.LC + B$ 
15   $R \leftarrow \text{updateRisk}(e_c)$ 

```

with ANNS as it returns approximate results. AKER enforces del-consistency that guarantees validity under deletions without requiring eager recomputation of the latest top- k , which would impose high refresh cost on the hit path.

Consider a cache that follows the del-consistency model. A cache entry for query q is created at time t_0 . If a cached neighbor v is deleted at t_1 ($t_1 > t_0$), the cache removes v immediately (P1). If a new vector u is inserted at t_2 ($t_2 > t_0$) and would appear in s_{t_2} , u may remain invisible to q 's cache entry until the next refresh at t_3 ($t_3 > t_2$). During this interval, the cache can still return a valid top- k (P2). AKER implements del-consistency by handling deletions immediately and insertions lazily, with a reserve parameter Δ .

Deletions. AKER immediately removes deleted vectors from the neighbor object pool to satisfy validity (P1). However, after removals, a cache entry may no longer contain k valid neighbors. In this case, the entry (cached query) must be invalidated. In AKER, a neighbor object can be referenced by multiple cache entries, thus a single vector deletion may invalidate multiple entries.

To reduce cache entry invalidations, AKER stores the top- k results with an additional Δ -sized reserve in NRL. We choose Δ to be small and collect the reserve using the same search path as for the top- k . ANNS algorithms typically maintain an internal candidate set that is larger than k during search, even though they return only the top- k results [18, 22, 43, 44]. Thus, AKER can save the $(k + \Delta)$ neighbors from the same traversal with negligible additional overhead. When a deletion invalidates a neighbor object, AKER removes it immediately and promotes the next closest reserved neighbor in NRL (P1, P2). If deletions break NRL such that the number of valid neighbor objects falls below k , AKER invalidates the corresponding cache entry.

Insertions. When a new vector is inserted into the database, it is unknown which cache entries should include it in their NRL. A naive approach that scans every cached query and updates affected NRLs is expensive. To minimize overhead, AKER splits insertion handling into a *fast path* and a *slow path* (Algorithm 2).

The fast path targets the cache entry that is most likely to be affected by the insertion (P2). Upon receiving an insert request, AKER appends the new vector to the insert log and queries the query

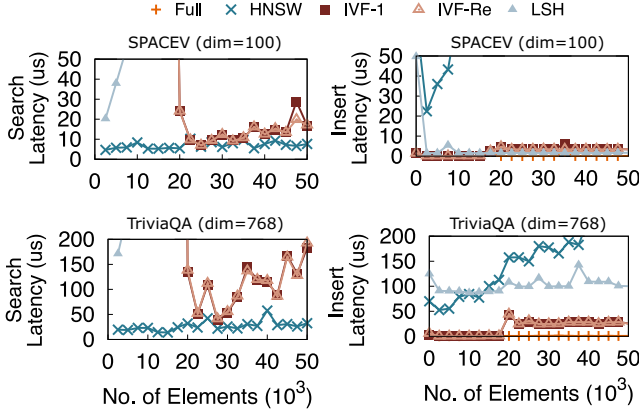


Figure 6: ANN index operation latency. We grow indexes to measure latency of top-1 lookup and incremental builds. We compare FAISS [26] HNSW ($m = 4$, $efConstruction = 8$, $efSearch = 16$), LSH ($nbits$ matched to vector dimensionality), IVF with single build at 20K inserts (IVF-1; IVFFlat $nlists = 128$, $nprobe = 1$), and with periodic rebuild at every 10K inserts (IVF-Re), and a brute-force scan (Full).

filter (line 3) to retrieve the closest representative entry. AKER then compares the distance between the inserted vector and the entry’s RQ against the entry’s $maxD$ (line 5). If the inserted vector is closer than $maxD$, AKER evicts the farthest neighbor from NRL, inserts a reference to the new vector, and updates $maxD$ accordingly.

Because the fast path updates only one entry, other entries that should also include the new vector may remain stale. To propagate insertions beyond the fast path target, the slow path incrementally refreshes the remaining entries (line 8). On each cache lookup or insert log append, AKER selects one representative entry in a round-robin manner (line 10). For the selected entry, AKER scans the next batch of log records (default 16) starting from its LC. For each scan, AKER computes the distance between the inserted vector and RQ (line 12) and substitutes the inserted vector in the same manner as the fast path. Then, AKER advances LC to the most recently examined log position and recomputes the global $risk$ R .

During vector insertions and cache lookups, AKER triggers the slow path when R exceeds a predefined risk threshold. AKER maintains R as a conservative estimate of potential recall degradation under delayed insert propagation. R is derived from per-entry *risk factors* r and the number of unchecked vectors in the insert log. For each representative entry e , $r(e)$ estimates how likely an unseen inserted vector could enter e ’s top- k by replacing a member of NRL. AKER uses the Δ reserve region in NRL as a proxy for how tight the top- k boundary is: since only the top- k results are returned, refreshing items at ranks $k+1$ to $k+\Delta$ can be delayed without changing the returned results (P2).

$$r(e) = \frac{d_k(e)}{d_{k+\Delta}(e)} \in [0, 1], \quad R = \bar{r} \cdot \frac{\bar{U}}{W} \quad (2)$$

Here, $d_\alpha(e)$ denotes the distance between e ’s query and the α -th neighbor in e ’s cached top- $(k+\Delta)$ list. A value of $r(e)$ close to 1 indicates a higher likelihood that an unseen insertion could substitute into the top- k . By design, AKER over-estimates the risk by treating all unchecked insert log records as potentially harmful.

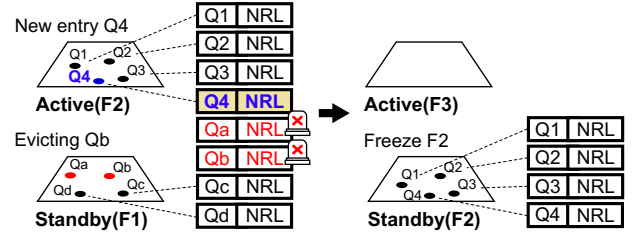


Figure 7: Query filter eviction.

$\bar{r}$ is the average risk factor, $\bar{U}$ is the average number of unchecked insert log records per cache entry, and W is the total insert log size. R scales with both how tight the top- k boundaries are and how far entries lag behind the insert log. As R increases toward 1, the potential for recall degradation increases.

Insert log. AKER reclaims insert log space when evicting a cache entry whose LC references the log head. Starting from the head, AKER advances the head and removes log records that are no longer referenced by the LC of any cache entry. This reclamation stops at the earliest log record that is referenced by at least one cache entry.

4.7 Eviction

Neighbor object pool. AKER evicts neighbor objects when a new cache entry requires allocating new neighbor objects but the pool has no free space. To free space, AKER selects a victim representative entry and reclaims the neighbor objects referenced by its NRL. We adopt FIFO because it has low overhead. Prior work suggests that under tight similarity thresholds, the accuracy impact of the eviction policy is minor [18]. Given a victim representative entry, AKER traverses its NRL and decrements the reference count of each neighbor object. If the reference count reaches zero, it is removed. Otherwise, the object remains alive because it is still referenced by other cache entries. If reclaiming a single victim entry does not provide enough free space, AKER evicts the next FIFO victim and continues until the space becomes sufficient.

When evicting a representative entry, AKER also removes all linked alias entries. It follows the representative’s next pointers, removes each alias entry, and deletes the corresponding query lookup table records. After all aliases are removed, AKER deletes the victim representative entry and removes it from the query lookup table. Finally, AKER marks the representative entry as a tombstone in the query filter to prevent it from being returned by subsequent approximate lookups.

Query filter. The query filter is the embedded ANN index over representative cached queries. To choose the best candidate for the query filter, we microbenchmark several index types [26] under an in-memory setting using two query sets (Section 5.1) as shown in Figure 6. The results show a clear trade-off between lookup and insertion latency across the index types. HNSW provides the lowest lookup latency even at higher dimensionality, but its insertion latency grows with the number of indexed elements. IVF shows low latency in insertion, but its lookup latency is higher than HNSW and grows more with the number of indexed elements. LSH is slowest in lookup and is less aligned with the distance metric used by the underlying ANNS. Because AKER targets read path optimization, we choose HNSW because it provides the lowest lookup latency.

The adoption of HNSW for the query filter introduces two major challenges. First, to keep insertion overhead of the query filter low, AKER must bound the size of the index. Second, graph-based ANN indexes like HNSW make fine-grained deletions non-trivial because the removal may break the graph connectivity [26, 32, 60]. Prior systems often implement deletions via tombstones and periodic rebuilds [32, 60]. This approach has three limitations. First, searches may return deleted items before the next rebuild. Second, tombstoned elements inflate the index size and increase insertion latency. Finally, selecting the timing of index rebuilds becomes important, since rebuilding from scratch can amplify tail latency.

To avoid per-entry deletions while bounding the size of the query filter, AKER implements a dual-index design where it decouples insertions from evictions and reclaims space by dropping an entire index at epoch boundaries. AKER maintains an *active filter* F_A , *standby filter* F_S , and a monotonically increasing epoch identifier. During an epoch, F_A admits new representative entries, whereas F_S admits no inserts and is used to isolate evictions. AKER advances the epoch when the number of evicted entries exceeds the number of live entries in the query filter. At an epoch transition, the filters swap roles, as shown in Figure 7. Under FIFO, the entries are evicted only from F_S (F1), and the evicted entries are marked as tombstones. When the number of tombstones exceeds the number of live entries in the query filter, AKER halts inserts into F_A (F2), seals it, and promotes it to become the next F_S . Then, AKER creates an empty index as next F_A (F3) and drops the previous F_S (F1), thereby reclaiming space without per-entry deletions or clean rebuilds. During lookup, AKER queries both F_A and F_S and selects the closest candidate cache entry. Because evictions never occur in F_A , any candidate returned from F_A is valid. Consequently, AKER guarantees the return of valid cache entries, never returning tombstoned entries.

Implementation. The current prototype uses coarse-grained synchronization through a global lock, as shared references require coordination across AKER components. We leave a fine-grained concurrent design to future work.

5 EVALUATION

We try to answer the following questions:

- Does the per-cached query similarity threshold maintain stable accuracy across different request distributions compared to single-threshold methods? (Section 5.2)
- Can AKER reduce read amplification compared to a traversal-centric cache? (Section 5.3)
- Can AKER sustain high performance without degrading recall under cache refreshes? (Section 5.4)

5.1 Experiment Setup

We integrate AKER into pgvector [9] HNSW (v0.8.0), a PostgreSQL extension, as our primary ANNS system. pgvector provides a page-based traversal-centric cache baseline via PostgreSQL shared buffer. We run all experiments on a server equipped with a four-socket Intel Xeon Platinum 8276 CPU, 512 GB RAM, and a 512 GB NVMe SSD, running Ubuntu 24.04. The client and PostgreSQL execute on the same machine. The client issues requests from a single thread. Unless otherwise stated, the default AKER uses $(\alpha_T, \alpha_L) = (0.25, 1.1)$, a query filter of $m = 4$, $efConstruction = 8$, $efSearch =$

Table 2: Build configurations and data sizes

Dataset	SPACEV(10M)	SPHERE(10M)	SPACEV(1M)
Dimension	100	768	100
pgvector	m=16, efCon=128, Vector 4.6 GB, Index 7.4 GB	m=32, efCon=256, Vector 41.0 GB, Index 40.9 GB	m=16, efCon=64, Vector 477.9 MB, Index 781.8 MB
DiskANN	R=64, L=100, On-disk 3.5 GB, In-mem 1.1 GB	R=96, L=100, On-disk 39.0 GB, In-mem 5.1 GB	-

16, and no deletion reserve ($\Delta = 0$). Section 5.4 evaluates variants derived from the default configuration.

PostgreSQL caches 8KB table and index pages in its shared buffer [12]. pgvector stores page-organized vectors and indexes as PostgreSQL tuples. These pages are also cached via the shared buffer, which cannot be disabled manually. To reduce the impact of buffers, we set the size to the minimum 128KB when evaluating approximate caches. pgvector can benefit from both shared buffers and OS cache [13]. Thus, we choose two settings for search workloads: a 16 GB *memory-constrained deployment* via container, and a *host deployment* where pgvector runs directly on the host. The memory-constrained deployment limits the OS caching support.

We also evaluate whether AKER improves searches on a highly optimized disk-based index, DiskANN [32]. We add AKER as a front-end caching layer for DiskANN. DiskANN already manages its own memory-resident index, so AKER forwards a query to DiskANN only on a cache miss. The benchmark runs with a single thread.

Datasets. We evaluate two application scenarios – web search and RAG – using SPACEV [11] and SPHERE [50] as summarized in Table 2. For RAG queries, we use TriviaQA [35]. For the search workload, we use 10M-scale datasets; for the refresh stress test, we use a 1M-scale dataset. We use a smaller scale in the refresh test to increase the likelihood that newly inserted vectors appear in the top- k , thereby stressing the refresh behavior. We build all indexes using Euclidean distance. For each workload, we set search parameters to achieve 0.9 recall at $k=10$ without approximate caches.

Queries. Real-world workloads are often skewed [6, 15, 18]. However, the query sets [5, 8, 11] are provided as sampled collections that lack request patterns. Because real traces are unavailable, we synthesize a workload called *simZipf* that jointly models query similarity and request skew.

We construct *simZipf* in two stages. First, for each base query, we build a similar-query group containing 50 queries. We take two different synthesis methods because each dataset exposes different forms of query information. For SPACEV, which provides only a query set Q of 29K vectors, we construct nearby queries directly in the vector space via linear interpolation. For each base vector $q_i \in Q$, we sample another vector $p \in Q$ and generate $q'_i = (1 - \lambda)q_i + \lambda p$, where λ is sampled uniformly from $(0, 0.5)$. Repeating this procedure with 50 sampled choices of p yields the similar-query group. For TriviaQA, which provides 10K raw natural-language questions, we generate 50 semantically similar questions using Mistral-7B [33]. Then, we encode the generated questions with the BGE-large encoder [59] to form the similar-query group.

Next, we generate a skewed request sequence over the similar-query groups using a Zipfian distribution [15, 18, 24, 61]. We use

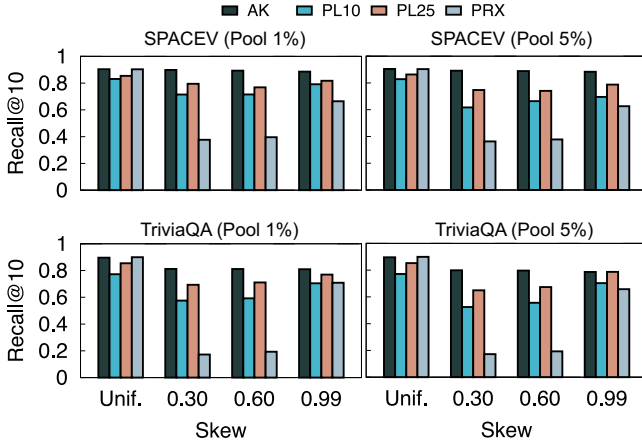


Figure 8: Recall. AKER (AK) maintains higher recall across skew levels than Potluck+ (PL) and PROXIMITY+ (PRX).

YCSB-C [6] to generate a 100K Zipfian key sequence ($k_1, k_2, \dots$), and map each similar-query group to the key with the same popularity rank. For each key access, we uniformly sample one query from the corresponding similar-query group. We evaluate three Zipfian exponents: 0.30, 0.60, and 0.99. We also add a uniform distribution using unmodified queries to provide a lowest locality reference.

Baselines. We compare AKER against Potluck and PROXIMITY, which are the latest approximate caches. We also compare against PostgreSQL’s default traversal-centric cache, the shared buffer. For a fair comparison, we implement PROXIMITY and Potluck within AKER by substituting only the threshold control module and keeping the rest of the system unchanged. All baselines share AKER’s neighbor object pool, NRL layout, and eviction policy. The differences reflect only the cache-hit rule. In both baselines, alias entries are disabled as they are AKER-specific optimizations.

- **Potluck+** [30]. We implement Potluck’s threshold-tuning mechanism. Following the Potluck algorithm, upon cache-entry insertion, we test set equivalence using the hashed vectors stored in NRL. At lookup time, Potluck performs dropout: with probability p , it forces a cache miss. We use Potluck’s default setting of $p=0.10$ and add a more aggressive setting of $p=0.25$.
- **PROXIMITY+** [18]. We use a strict threshold for high recall. We measure all inter-query distances between the synthesized queries and use the 1st-percentile value as the fixed similarity threshold.

5.2 Cache Accuracy

We evaluate all cache methods at $k = 10$. This is a strict criterion, as each single missing ground truth neighbor reduces recall by 0.1. We target at least 0.8 recall for AKER, corresponding to fewer than one missed true neighbor per query. We set the neighbor object pool to store up to 1% and 5% of the dataset vectors. AKER shows consistent trends in recall and hit ratios across pgvector and DiskANN; thus we provide results for pgvector in Figure 8 and 9.

Accuracy. Across varying skewness, AKER achieves up to 52.3 percentage points (pp) and 64.0 pp higher recall than the single similarity threshold methods on SPACEV and TriviaQA, respectively. In all settings, AKER misses fewer than one correct result

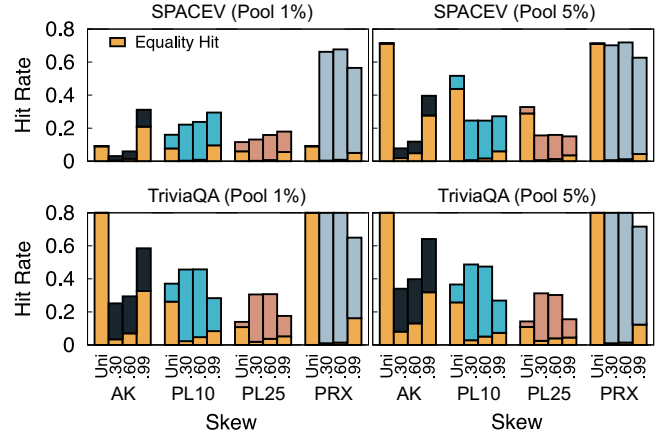


Figure 9: Hit ratios. Each stacked bar shows the total cache hit ratio. The yellow segment denotes equality hits, and the remaining segment denotes approximate hits.

on average and maintains recall above 0.80. The single similarity thresholds become overly permissive, producing more approximate hits. The low recall persists even when PROXIMITY+ (PRX) uses a conservative setting of 1% inter-query distance, indicating that a strict constraint is insufficient to preserve accuracy.

Potluck+ (PL) achieves higher recall than PRX by forcing pgvector searches. Higher p leads to a higher recall. With $p = 0.1$ (PL10), the maximum recall reaches 0.79 and 0.71 on each dataset, still missing more than one correct item. Higher recall is achieved by more frequently invoking the ANNS. Increasing p to 0.25 (PL25) improves recall compared to PL10, but both still remain worse than AKER. Under low skew, query distributions become diverse. A single threshold fails to capture query-specific density. Thus, as skew decreases, the recall gap between AKER and other methods widens.

Cache hit ratios. Figure 9 shows the hit ratios under varying skewness levels (x -axis) and cache sizes. As skew increases, near-duplicate queries recur more frequently; all caches show increased total hit ratios. PRX achieves high hit ratios across all configurations, but mostly via approximate hits, which substantially reduce recall. PL also shows a higher approximate hit ratio than AKER. Although a larger p triggers more ANNS, PL’s reliance on a single similarity threshold produces high approximate hits. In contrast, AKER adapts to workload skewness. As skew grows, AKER relies less on approximate hits. AKER shows higher recall because multiple thresholds allow approximate hits only when they are locally safe.

The uniform distribution further highlights the difference in the threshold control. As queries recur, equality hits dominate in AKER and the recall is preserved. In PL, the equality hits gradually loosen its global threshold and allows farther approximate matches.

5.3 Cache Efficiency

Operational overhead. Figure 10 shows that AKER adds little overhead compared to pgvector processing latency. The average cache lookup latency remains below 0.1 ms. A cache miss adds less than 1 ms, from cache lookup and cache entry insertion combined.

Read amplification. We observe read amplification by measuring the average number of PostgreSQL 8 KB pages read per query

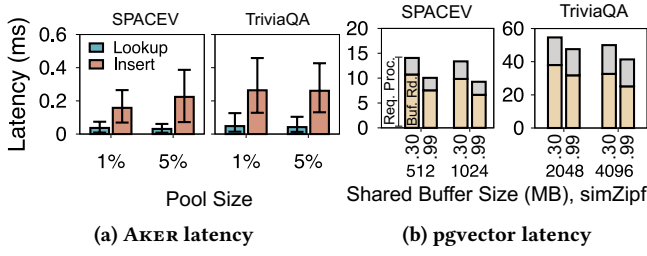


Figure 10: AKER (lookup and cache entry insert) and pgvector latency. Each bar shows the average, and error bar shows 1st- and 99th-percentile latencies.

from PostgreSQL logs. We compare AKER (AK) against pgvector’s traversal-centric cache, PostgreSQL shared buffer (PG). For AKER, we set the maximum neighbor object pool size to 1%, 2%, and 5%. For the shared buffer, we set the size from 128 MB to 8 GB.

Figure 11 shows that AKER reduces read amplification, ρ , on both datasets. Under high skew (0.99), AKER reduces ρ to 0.7 $\times$ and 0.4 $\times$ of pgvector on SPACEV and TriviaQA, while using only 0.4 $\times$ and 0.5 $\times$ of shared buffer memory, respectively. Under low skew (0.30), AKER still achieves 1.0 $\times$ and 0.7 $\times$ ρ in both datasets while using 0.8 $\times$ and 0.6 $\times$ of pgvector’s memory. As ANNS requires large reads, each query can still read many pages when the working set exceeds the buffer. In AKER, page reads occur only on cache misses.

QPS. Figure 12a reports QPS under two pgvector deployment settings. On SPACEV, the 16 GB limit shows little difference in QPS as the data size fits the memory budget. On TriviaQA, in contrast, QPS decreases because the data is larger than the memory budget. In all settings, AKER achieves up to 2 $\times$ and 3 $\times$ QPS while using 0.8 $\times$ and 0.6 $\times$ the memory on SPACEV and TriviaQA, respectively.

Despite DiskANN’s strong baseline efficiency, AKER provides improvements, with benefits varying across datasets and workload skew. On 100-dimensional SPACEV, DiskANN search is inexpensive, achieving an average query latency of 3.7 ms, with 70% spent on reads and 30% on index traversal. Under lower-skew workloads, the hit rate is not high enough to amortize AKER’s lookup cost shown in Figure 10. Thus, QPS improves only at simZipf 0.99. On 768-dimensional TriviaQA, DiskANN search is substantially more expensive: the average query latency rises to 21.1 ms, with 31% spent on reads and 67% on index traversal. Although this latency is lower than that of pgvector, the higher backend search cost makes

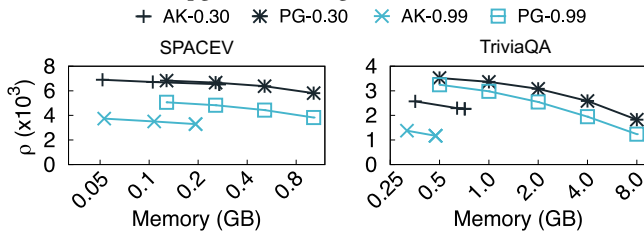
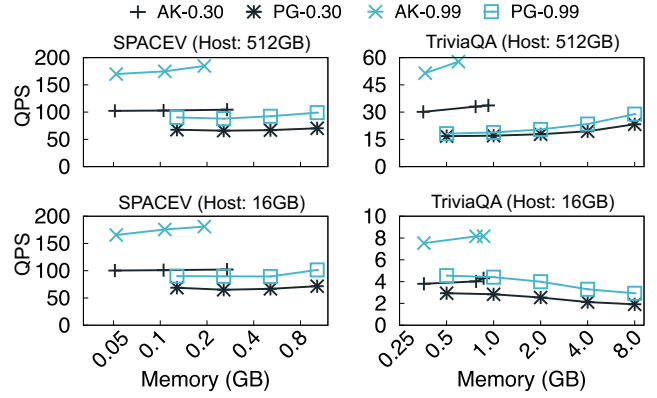
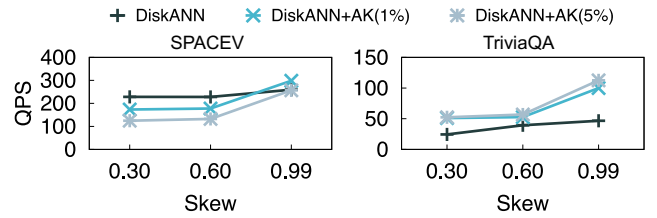


Figure 11: Read amplification. Numeric labels show the skewness. We omit simZipf 0.60 as it follows the same trend as simZipf 0.30. The memory use of the query filter is approximated as: (1) R query vectors of cache entries (dimension d , float32) are $R \cdot d \cdot 4$ bytes, (2) maximum number of neighbors $m = 4$, storing m 8-byte identifiers per entry are $R \cdot m \cdot 8$ bytes.



(a) QPS of pgvector.



(b) QPS of DiskANN. The additional memory footprint of AKER over DiskANN remains small, reaching at most 26.4 MB on SPACEV and 275.6 MB on TriviaQA, which corresponds to 2.5% and 5.4% of DiskANN’s pinned in-memory index, respectively.

Figure 12: QPS. Figure 12a and Figure 12b follow the same approximation and plotting convention as in Figure 11.

bypassing the full search path more beneficial. Thus, AKER achieves up to 2.4 $\times$ higher QPS across all skews on TriviaQA.

Memory efficiency. Figure 13 shows NRL lengths and the query filter’s share of total memory in simZipf 0.99. An NRL length is the ratio of the number of neighbor objects to the number of cache entries. The NRL length below $k = 10$ indicates neighbor objects are shared across entries. Compared to a naive design that allocates k neighbor objects per entry, AKER reduces the number of neighbor objects needed by 2.9-6.1 $\times$. Figure 13b shows that the number of cache entries brings minor overhead, with the query filter accounting for only 13.5-18.8% of total memory.

5.4 Cache Refresh

To invoke cache refresh, we first make the cache dirty, then run searches on the AKER-attached pgvector. Using a 1% neighbor object

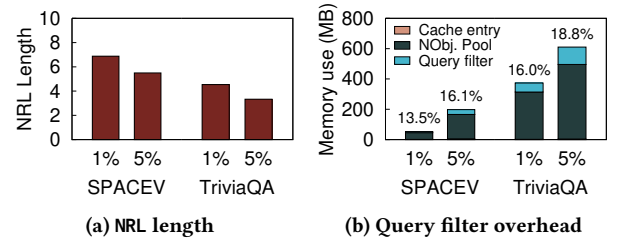


Figure 13: Memory efficiency. Figure 13b shows the memory usage, following the same approximation of Figure 12a.

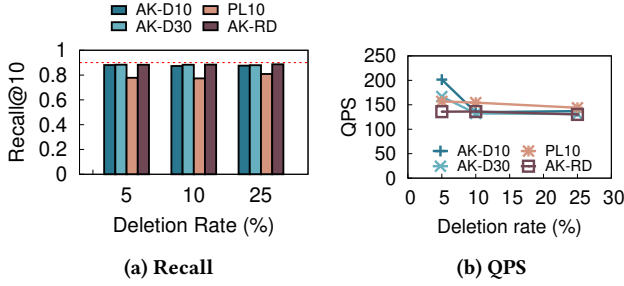


Figure 14: Refresh stress-test under updates. The red line is the recall when no approximate cache is attached. AKER (AK) maintains high recall and QPS compared to Potluck+ (PL) and the refresh-disabled variant of AKER (RD).

pool, we first warm up the cache by running simZipf 0.99. Next, we insert an additional 5% of vectors to fill the insert log from another split of SPACEV [60]. We then randomly invalidate neighbor objects to simulate deletions. We use deletion levels of 5%, 10% and 25%. Finally, the same search queries are re-executed.

For AKER, we set Δ to 10 (AK-D10) and 30 (AK-D30), the risk threshold to 0.3, and the slow path batch size B to 16. We compare against Potluck+ with probability $p = 0.10$ (PL10). We include a no-refresh ablation with $\Delta = 0$, AK-RD, in which refresh mechanisms are disabled in AKER. For PL10 and AK-RD, no refresh is performed. Under deletions, if a cache entry cannot maintain k valid results, it is treated as a miss to avoid correctness violations (P1).

Accuracy and QPS. AK-D10 and AK-D30 incur at most 2 pp recall drop across all deletion rates. AK-RD shows the highest recall because deletions invalidate many entries and trigger the ANNS more frequently. In Table 3, AK-RD shows a low cache hit ratio of 6.4% due to the large number of invalidated cache entries. In contrast, PL10 suffers from a large recall drop of 12.3 pp, even at low deletion rates. At a deletion rate of 5%, AK-D10 achieves the highest QPS, outperforming PL10 by 1.3 $\times$ and AK-RD by 1.5 $\times$.

Selection of Δ . Under a fixed pool budget, increasing Δ improves per-entry tolerance to deletions (more reserves per cached query), but each cache entry consumes more neighbor objects. Thus, the number of cache entries decreases. Table 3 illustrates the effect. AK-D10 maintains 2 $\times$ more cache entries with a higher hit ratio than AK-D30. Because AK-RD cannot recover cache entries, it shows the lowest hit ratio and QPS, even though it maintains 4 $\times$ more cache entries than AK-D30. Overall, AK-D10 achieves the best deletion tolerance-recall balance, including the most aggressive stress level up to 25% deletions. This suggests that a small Δ is sufficient to tolerate the deletion pressure.

Refresh overhead. Figure 15 reports the refresh latency in AK-D10 and compares it against the end-to-end search latency of pgvector. The log insert latency includes both the fast and slow paths. Overall, the refresh overhead remains small, never exceeding 0.25 ms. On

Table 3: Number of entries and hit ratio at 5% deletion rate.

	AK-D10	AK-D30	PL10	AK-RD
# of entries	625	315	1,170	1,284
Hit ratio	12.52%	9.71%	16.64%	6.39%

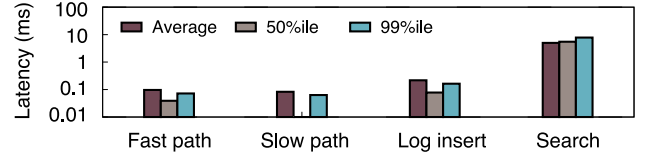


Figure 15: Cache refresh overhead (AK-D10).

average, AKER adds only 0.22 ms, which is about 4.4% of the 5 ms search latency. The results indicate that AKER introduces negligible refresh overhead relative to query processing.

6 RELATED WORK

ANN index optimizations. These studies propose optimized ANN index designs that reduce disk I/O and memory costs. GoVector [65] proposes an index layout that places similar vectors on the adjacent disk pages to reduce I/O. DiskANN [32] and SPANN [22] propose a memory-disk hybrid index design to overcome extensive memory use. DiskANN proposes the Vamana algorithm and a hybrid storage layout that keeps quantized vectors in-memory and full-precision vectors on-disk. SPANN proposes a balanced clustering algorithm to reduce disk I/O overhead. SPFresh [60] extends SPANN to support incremental index builds while maintaining high accuracy under vector updates. CXL-ANNS [31] and SmartANNS [54] propose hardware-optimized index designs to accelerate index traversal.

Approximate caches. These studies introduce approximate caching across diverse domains. Prior works have used approximate caches in systems for advertising [45], image retrieval [28], and scene understanding [49]. Potluck [30] proposed function-level caching to reduce cross-application computation. RAGCache [34] caches frequently referenced knowledge chunks to reduce end-to-end response time in RAG pipelines. GPTCache [17] and PROXIMITY [18] leverage query similarity to reduce redundant document searches. These systems do not address query-specific reuse control or cache refreshes, which are the key designs of AKER.

7 CONCLUSION

We present AKER, an approximate cache for disk-based ANNS. AKER targets two challenges in approximate caching: maintaining high recall and throughput, and designing a low-cost cache refresh mechanism. First, AKER introduces per-query similarity thresholds that tailor hit decisions to each cached query based on observed hit behavior, preserving recall across varying neighborhood densities. Second, AKER proposes del-consistency, which applies deletions eagerly while handling insertions lazily via Δ reserves and risk scores that enable a low-overhead cache refresh pipeline. We integrate AKER into pgvector. AKER improves recall by up to 64 percentage points over prior solutions and increases QPS by up to 3.2 $\times$ relative to PostgreSQL’s shared buffer, while using only 0.6 $\times$ of the memory.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments and suggestions. This research was supported by the IITP-MSIT (Institute for Information & Communications Technology Planning & Evaluation, Ministry of Science and ICT) (RS-2024-00436680 and RS-2024-00459026), Microsoft Research, and NRF (National Research Foundation of Korea) (RS-2025-00518369).

REFERENCES

- [1] [n.d.]. Amazon RDS for PostgreSQL. <https://aws.amazon.com/rds/postgresql/>. Accessed: 2025-06-13.
- [2] [n.d.]. Google Cloud Documentation: Generate and manage vector embeddings. <https://docs.cloud.google.com/sql/docs/postgres/generate-manage-vector-embeddings>. Accessed: 2025-06-13.
- [3] [n.d.]. pgvector in Azure Database - Flexible Server. <https://learn.microsoft.com/en-us/azure/postgresql/flexible-server/how-to-use-pgvector>. Accessed: 2025-06-13.
- [4] [n.d.]. Redis. <https://redis.io>. Accessed: 2025-06-13.
- [5] 2010. Datasets for Approximate Nearest Neighbor Search. <http://corpus-texmex.irisa.fr>. Accessed: 2025-06-13.
- [6] 2014. YCSB-C. <https://github.com/basicthinker/YCSB-C>. Accessed: 2025-06-13.
- [7] 2020. Weaviate. <https://weaviate.io>. Accessed: 2025-06-13.
- [8] 2021. NeurIPS 2021 Competition Track. <https://neurips.cc/Conferences/2021/CompetitionTrack>. Accessed: 2025-06-13.
- [9] 2021. pgvector. <https://github.com/pgvector/pgvector>. Accessed: 2025-06-13.
- [10] 2021. Qdrant. <https://qdrant.tech>. Accessed: 2025-06-13.
- [11] 2021. SPACEV1B: A Billion-scale Vector Dataset for Text Descriptors. <https://github.com/microsoft/SPTAG/tree/master/datasets/SPACEV1B>. Accessed: 2025-06-13.
- [12] 2023. PostgreSQL Documentation. <https://www.postgresql.org/docs/16/>. Accessed: 2025-06-13.
- [13] 2025. PostgreSQL Documentation: Resource Consumption. <https://www.postgresql.org/docs/current/runtime-config-resource.html>. Accessed: 2025-06-13.
- [14] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altmenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2023).
- [15] Lada A Adamic and Bernardo A Huberman. 2002. Zipf's Law and the Internet. *Glottometrics* 3, 1 (2002), 143–150.
- [16] Dávid Bajusz, Anita Rácz, and Károly Héberger. 2015. Why is Tanimoto index an appropriate choice for fingerprint-based similarity calculations? *Journal of Cheminformatics* 7, 1 (2015), 20.
- [17] Fu Bang. 2023. GPTCache: An Open-Source Semantic Cache for LLM Applications Enabling Faster Answers and Cost Savings. In *Proc. of the 3rd Workshop for Natural Language Processing Open Source Software (NLP-OSS)*. 212–218.
- [18] Shai Bergman, Anne-Marie Kermarrec, Diana Petrescu, Rafael Pires, Mathis Randl, Martijn De Vos, and Ji Zhang. 2025. Leveraging Approximate Caching for Faster Retrieval-Augmented Generation. In *Proc. of the 26th International Middleware Conference (MIDDLEWARE)*. 340–353.
- [19] B Barla Cambazoglu, Ismail Sengor Altinoglu, Rifat Ozcan, and Özgür Ulusoy. 2012. Cache-Based Query Processing for Search Engines. *ACM Transactions on the Web (TWEB)* 6, 4 (2012), 1–24.
- [20] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proc. of 18th USENIX Conference on File and Storage Technologies (FAST)*. 209–223.
- [21] Cheng Chen, Chenzhe Jin, Yunan Zhang, Sasha Podolsky, Chun Wu, Szu-Po Wang, Eric Hanson, Zhou Sun, Robert Walzer, and Jianguo Wang. 2024. SingleStore-V: An Integrated Vector Database System in SingleStore. *Proc. of the VLDB Endowment* 17, 12 (2024), 3772–3785.
- [22] 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.
- [23] Flavio Chierichetti, Ravi Kumar, and Sergei Vassilvitskii. 2009. Similarity Caching. In *Proc. of the twenty-eighth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*. 127–136.
- [24] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proc. of the 1st ACM symposium on Cloud Computing (SoCC)*. 143–154.
- [25] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for YouTube recommendations. In *Proc. of the 10th ACM Conference on Recommender Systems (RecSys)*. 191–198.
- [26] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2026. The Faiss Library. *IEEE Transactions on Big Data* 12 (2026), 346–361.
- [27] Fabrizio Falchi, Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fausto Rabitti. 2008. A Metric Cache for Similarity Search. In *Proc. of the 2008 ACM workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR)*. 43–50.
- [28] Fabrizio Falchi, Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fausto Rabitti. 2012. Similarity caching in large-scale image retrieval. *Information Processing & Management* 48, 5 (2012), 803–818.
- [29] Alessandro Finamore, James Roberts, Massimo Gallo, and Dario Rossi. 2022. Accelerating Deep Learning Classification with Error-controlled Approximate-key Caching. In *Proc. of IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 2118–2127.
- [30] Peizhen Guo and Wenjun Hu. 2018. Potluck: Cross-Application Approximate Deduplication for Computation-Intensive Mobile Applications. In *Proc. of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 271–284.
- [31] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. 2023. CXL-ANNS: Software-Hardware Collaborative Memory Disaggregation and Computation for Billion-Scale Approximate Nearest Neighbor Search. In *Proc. of 2023 USENIX Annual Technical Conference (ATC)*. 585–600.
- [32] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. *Advances in Neural Information Processing Systems* 32 (2019).
- [33] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Léo Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. *arXiv preprint arXiv:2310.06825* (2023).
- [34] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Shufan Liu, Xuanzhe Liu, and Xin Jin. 2025. RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation. *ACM Transactions on Computer Systems (TOCS)* 44, 1 (2025), 1–27.
- [35] Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. 2017. TriviaQA: A Large Scale Distantly Supervised Challenge Dataset for Reading Comprehension. In *Proc. of the 55th Annual Meeting of the Association for Computational Linguistics (ACL)*. 1601–1611.
- [36] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H Noh, and Young-ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *Proc. of 17th USENIX Conference on File and Storage Technologies (FAST)*. 191–205.
- [37] Noam Koenigstein, Parikshit Ram, and Yuval Shavitt. 2012. Efficient retrieval of recommendations in a matrix factorization framework. In *Proc. of the 21st ACM International Conference on Information and Knowledge Management (CIKM)*. 535–544.
- [38] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.
- [39] Hui Li, Tsz Nam Chan, Man Lung Yiu, and Nikos Mamoulis. 2017. FEXIPRO: Fast and Exact Inner Product Retrieval in Recommender Systems. In *Proc. of the 2017 ACM International Conference on Management of Data (SIGMOD)*. 835–850.
- [40] 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 *Proc. of the 19th International Middleware Conference Industry (MIDDLEWARE)*. 9–16.
- [41] Defu Lian, Haoyu Wang, Zheng Liu, Jianxun Lian, Enhong Chen, and Xing Xie. 2020. LightRec: A Memory and Search-Efficient Recommender System. In *Proc. of The Web Conference 2020 (WWW)*. 695–705.
- [42] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazheng Gu, Xin Wang, Michael R Lyu, and Yangfan Zhou. 2023. SMART: A High-Performance Adaptive Radix Tree for Disaggregated Memory. In *Proc. of 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 553–571.
- [43] Craig Macdonald and Nicola Tonellotto. 2021. On Approximate Nearest Neighbour Selection for Multi-Stage Dense Retrieval. In *Proc. of the 30th ACM International Conference on Information & Knowledge Management (CIKM)*. 3318–3322.
- [44] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE transactions on Pattern Analysis and Machine Intelligence* 42, 4 (2018), 824–836.
- [45] Sandeep Pandey, Andrei Broder, Flavio Chierichetti, Vanja Josifovski, Ravi Kumar, and Sergei Vassilvitskii. 2009. Nearest-Neighbor Caching for Content-Match Applications. In *Proc. of the 18th International Conference on World Wide Web (WWW)*. 441–450.
- [46] Jie Ren, Minjia Zhang, and Dong Li. 2020. HM-ANN: Efficient Billion-Point Nearest Neighbor Search on Heterogeneous Memory. *Advances in Neural Information Processing Systems* 33 (2020), 10672–10684.
- [47] Tareq Si Salem, Giovanni Neglia, and Damiano Carra. 2022. Ascent Similarity Caching with Approximate Indexes. *IEEE/ACM Transactions on Networking* 31, 3 (2022), 1173–1186.
- [48] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2021. Constructing and Analyzing the LSM Compaction Design Space. *Proc. of the VLDB Endowment* 14, 11 (2021), 2216–2229.
- [49] Surya Selvam, Ravi K. Rajendran, Murugan Sankaradas, Anand Raghunathan, and Srimat T. Chakradhar. 2025. SimCache: Similarity Caching for Efficient VLM-based Scene Understanding. In *Proc. of IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 3318–3327.
- [50] Michael Shen, Muhammad Umar, Kiwan Maeng, G Edward Suh, and Udit Gupta. 2025. Hermes: Algorithm-System Co-design for Efficient Retrieval-Augmented

- Generation At-Scale. In *Proc. of the 52nd Annual International Symposium on Computer Architecture (ISCA)*. 958–973.
- [51] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. 2021. FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search. *arXiv preprint arXiv:2105.09613* (2021).
- [52] Jaime Teevan, Eytan Adar, Rosie Jones, and Michael AS Potts. 2007. Information re-retrieval: repeat queries in Yahoo’s logs. In *Proc. of the 30th Annual International ACM SIGIR conference on Research and Development in Information Retrieval (SIGIR)*. 151–158.
- [53] Bing Tian, Haikun Liu, Zhuohui Duan, Xiaofei Liao, Hai Jin, and Yu Zhang. 2024. Scalable Billion-point Approximate Nearest Neighbor Search Using SmartSSDs. In *Proc. of 2024 USENIX Annual Technical Conference (ATC)*. 1135–1150.
- [54] Bing Tian, Haikun Liu, Yuhang Tang, Shihai Xiao, Zhuohui Duan, Xiaofei Liao, Hai Jin, Xuechang Zhang, Junhua Zhu, and Yu Zhang. 2025. Towards High-throughput and Low-latency Billion-scale Vector Search via CPU/GPU Collaborative Filtering and Re-ranking. In *Proc. of 23rd USENIX Conference on File and Storage Technologies (FAST)*. 171–185.
- [55] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xianguyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A Purpose-Built Vector Data Management System. In *Proc. of the 2021 International Conference on Management of Data (SIGMOD)*. 2614–2627.
- [56] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xianguyu 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. *Proc. of the ACM on Management of Data (PACMOD)* 2, 1 (2024), 1–27.
- [57] 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. *Proc. of the VLDB Endowment* 13, 12 (2020), 3152–3165.
- [58] Shitao Xiao, Zheng Liu, Weihao Han, Jianjin Zhang, Yingxia Shao, Defu Lian, Chaozhao Li, Hao Sun, Denvy Deng, Liangjie Zhang, et al. 2022. Progressively Optimized Bi-Granular Document Representation for Scalable Embedding Based Retrieval. In *Proc. of the ACM Web Conference 2022 (WWW)*. 286–296.
- [59] Shitao Xiao, Zheng Liu, Peitian Zhang, Niklas Muennighoff, Defu Lian, and Jian-Yun Nie. 2024. C-Pack: Packed Resources for General Chinese Embeddings. In *Proc. of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*. 641–649.
- [60] 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 *Proc. of the ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP)*. 545–561.
- [61] Juncheng Yang, Yao Yue, and KV Rashmi. 2021. A Large-scale Analysis of Hundreds of In-memory Key-value Cache Clusters at Twitter. *ACM Transactions on Storage (TOS)* 17, 3 (2021), 1–35.
- [62] Wen Yang, Tao Li, Gai Fang, and Hong Wei. 2020. PASE: PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension. In *Proc. of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2241–2253.
- [63] Xizhe Yin, Chao Gao, Zhijia Zhao, and Rajiv Gupta. 2025. PANNS: Enhancing Graph-based Approximate Nearest Neighbor Search through Recency-aware Construction and Parameterized Search. In *Proc. of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 369–381.
- [64] Qiang Zhang, Yongkun Li, Patrick PC Lee, Yinlong Xu, and Si Wu. 2022. DEPART: Replica Decoupling for Distributed Key-Value Storage. In *Proc. of 20th USENIX Conference on File and Storage Technologies (FAST)*. 397–412.
- [65] Yijie Zhou, Shengyuan Lin, Shufeng Gong, Song Yu, Shuhao Fan, Yanfeng Zhang, and Ge Yu. 2025. GoVector: An I/O-Efficient Caching Strategy for High-Dimensional Vector Nearest Neighbor Search. *arXiv preprint arXiv:2508.15694* (2025).