arXiv:2608.25185v1 [cs.DB] 25 Aug 2026

Analyzing and Reducing Search Quality Differences in Vector Similarity Search

Sara Mahdizadeh Shahri, Martin Prammer, Jignesh M. Patel, Akshitha Sriraman
Carnegie Mellon University

Abstract

Modern database services scalably search over large data collections via Approximate Nearest Neighbor Search, which improves search performance at the cost of search quality, measured by *recall*. In practice, a database operator seeks to achieve a target mean recall while maximizing throughput across search queries. We show that optimizing for mean recall masks significant differences in recall across queries even when target recall is met. As a result, numerous queries face (1) below-target recall, hurting user experience and revenue and (2) above-target recall, wasting computation to deliver unnecessarily high search quality. Thus, it is critical to detect and reduce recall differences across queries.

We design *RCheck*, a light-weight run-time system that identifies low-recall queries and reduces recall differences while achieving high throughput. *RCheck*'s key design principle is to dynamically, efficiently adapt search effort by increasing effort for queries below target recall and decreasing effort for those above it. *RCheck* tunes available search effort parameters, making it readily deployable. We evaluate *RCheck* using the widely-used production-style pgvector [6] database. At the same throughput, *RCheck* improves mean recall by 11–93% and enables 8–47% more queries to meet target recall compared to the state-of-the-art globally-tuned configuration.

1 Introduction

Modern data management services perform *embedding-based similarity* search over large collections of data items, such as text, images, and videos [12, 28, 29, 33, 37, 42, 46, 59, 62, 66]. To enable such search, data items are encoded as high-dimensional *embedding vectors* [38, 47, 50] and stored in a *vector database* [6, 8, 24, 61]. These vectors form an *embedding space*, where nearby vectors map to similar items [38, 47, 50].

To search similar items for a query, the service encodes the query as an embedding vector and finds the closest stored vectors, i.e., nearest neighbors. In principle, we can find nearest neighbors by comparing the query vector against all stored vectors in the database. But, such exhaustive search is computationally expensive at modern scales [38, 55].

In practice, to enable scalable search, vector databases construct index structures that organize vectors via graphs, trees, hashmaps, or partitions [16, 35, 45, 51, 52, 60]. These indices enable Approximate Nearest Neighbor Search (ANNS), which searches only a subset of stored vectors. ANNS is often controlled by a *global search parameter* that governs *search effort*, i.e., the number of stored vectors scanned per query. Thus, ANNS improves search performance while potentially missing some true nearest neighbors, compromising search quality [16, 31, 32, 55]. Search quality loss is measured by *recall*, defined as the fraction of true nearest neighbors retrieved; recall closer to 1 indicates better search quality [15].

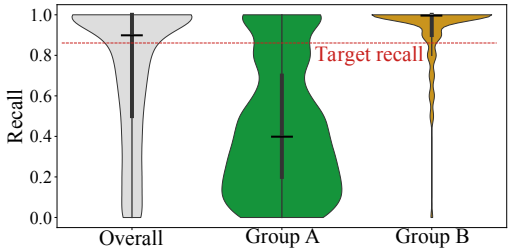


Figure 1. Recall distribution across all queries and within two query groups corresponding to different regions of the embedding space. Although the overall mean recall meets the target, recall varies substantially across groups, indicating uneven search quality across queries.

Typically, database operators tune indices to achieve a target mean recall (e.g., 0.85) while maximizing throughput across queries [8, 55, 63]. We find that optimizing for mean recall can mask a critical problem: search quality can differ substantially across queries even when target recall is met.

To illustrate recall differences across queries, Fig. 1 shows overall recall distribution for a *pgvector* database [6] with a graph-based index [51] built on a Twitter dataset [49] with GloVe-100 embeddings [50]. We split the dataset into index and query sets. While overall mean recall of 0.87 exceeds the 0.85 target, the distribution is wide rather than concentrated, indicating significant difference in recall across queries.

To better study recall differences across queries, we organize stored vectors into *groups* based on their location in the embedding space using popular k-means clustering [2]. We show that computing such groups offline, independent of the index structure, provides a logical characterization of the embedding space without modifying the index, which enables efficiently studying recall differences across queries.

Fig. 1 shows significant recall differences among real-world Twitter queries [49] that naturally map to different groups based on their embeddings: queries in group A average 0.21 recall while those in B average 0.99, despite similar cluster sizes. Thus, even when net mean recall meets the target, search quality can differ greatly across query groups. We show that recall differences persist across index configurations, index types, and embedding models [43, 57].

Such recall differences have adverse consequences. Queries that belong to low-recall groups experience poor search quality, which hurts (1) user experience, increasing user

abandonment and reducing engagement [10, 13, 40, 64] and (2) user trust and reliable service operation, hurting revenue [21, 23, 27, 63]. Thus, it is crucial to detect and reduce recall differences across queries.

Reducing recall differences is challenging. One option is to modify the index offline (e.g., by increasing graph connectivity for low-recall groups). But, such approaches are expensive and disruptive, making them challenging to deploy in production [45]. An alternative is to configure the global search parameter that controls *search effort* [6, 51, 60]. In current practice, this parameter is set coarsely and the same search effort is applied across all queries. Thus, increasing this parameter improves recall for queries in low-recall groups (e.g., group A), but, also increases search effort for queries that already meet the target recall, reducing net throughput.

Our key insight is that we can reduce recall differences while achieving high throughput by tuning search effort at a finer granularity, i.e., increasing effort for groups below target recall and decreasing it for those above it. Such tuning "takes from the rich and gives to the poor" by reallocating computation from high-recall groups to low-recall groups, reducing recall differences while ensuring high throughput.

Using this insight, we design *RCheck*, a lightweight run-time system that (1) identifies low-recall queries and (2) reduces recall differences while achieving high throughput. *RCheck* introduces several novel design principles. First, as computing recall differences across individual queries is expensive at modern scales, *RCheck* efficiently captures such differences by introducing *per-group* recall measurement. Second, unlike the modern coarse-grained practice of applying uniform search effort across queries, *RCheck* dynamically adapts effort *per-group* in a way that reduces recall difference while ensuring high throughput. Third, for quick adaptation, *RCheck* includes control-loop feedback to monitor recall and performance metrics and suitably adapt search effort at run time. Fourth, *RCheck* tunes available search effort parameters, requiring no expensive, intrusive index changes, making it index independent and readily deployable.

We evaluate *RCheck* using pgvector [6], a widely-used PostgreSQL extension with production-style ANNS [3–5, 15]. Across multiple datasets and index types, *RCheck* consistently improves recall-throughput trade-off. At the same target recall, *RCheck* improves throughput by 7–41% compared to the best globally-tuned configuration. At the same throughput, *RCheck* tightens the recall distribution, improving mean recall by 11–93% and enabling 8–47% more queries to meet target recall target.

In summary, we contribute:

- A demonstration that ANN recall can differ significantly across queries, despite meeting the target recall.
- A characterization of this recall difference across index types, index configurations, and embedding models.
- *RCheck*: A light-weight run-time system that reduces recall differences across queries without hurting throughput.
- A detailed evaluation of *RCheck*, demonstrating that it shifts the recall-performance tradeoff closer to the ideal.

2 Background

Vector databases, such as pgvector [6], FAISS [24], Milvus [61], and Weaviate [8] store embedding vectors at scale [55]. For a query vecto r, the database retrieves most-similar stored vectors under a distance metric (e.g., Euclidean distance) [32, 55].

Exact similarity search compares the query vector against all stored vectors, which is computationally infeasible at modern scales [22, 32, 55]. To improve scalability, these databases use Approximate Nearest Neighbor Search (ANNS) [32], which trades search accuracy for lower latency and higher throughput. ANNS stores vectors using index structures such as graphs, trees, hashmaps, or partitions, enabling searching only a subset of stored vectors for a query [16, 35, 45, 51, 52, 60]. Thus, ANNS may miss some true nearest neighbors, hurting search quality, which is quantified using *recall*, the fraction of true neighbors retrieved [15]. In practice, operators configure index parameters to achieve a target *mean recall* while maximizing throughput [8, 15, 55, 63].

Widely-used ANNS indexes include Hierarchical Navigable Small Worlds (HNSW) and IVFFlat. HNSW organizes vectors as a multi-layer graph which it greedily traverses to find nearest neighbors [45, 51]. IVFFlat partitions vectors into clusters and searches clusters closest to the query [60]. These ANNS techniques expose tunable parameters that govern index construction and run-time search effort, which together determine search quality vs. throughput tradeoff [51, 60].

In HNSW, build-time parameters determine the graph structure: `M` controls the number of connections per stored vector and `ef_construction` controls how many candidates are considered when selecting these connections during insertion. At run-time, `ef_search` determines how many stored vectors are scanned per query. In IVFFlat, `nlist` dictates the number of partitions created during index construction. `nprobes` controls how many partitions are searched at run-time [51, 60]. Increasing these parameters typically improves recall by exploring the embedding space more at the cost of higher latency and lower throughput (in Queries Per Second—QPS); decreasing them has the opposite effect [15].

Build-time parameters (`M`, `ef_construction`, `nlist`) require index reconstruction to modify. Run-time parameters which determine *search effort* (`ef_search` and `nprobes`) can be adjusted dynamically without index reconstruction. In practice, such run-time parameters are typically configured globally and applied uniformly across all queries [15, 24, 61].

3 Studying Recall Differences Across Queries to a Vector Database

We demonstrate recall differences across queries to a vector database, motivating why it is key to reduce such differences.

We show that such differences persist even when we change the embedding model, index type, index construction, and global search effort, motivating our solution, *RCheck*.

Demonstrating recall differences across queries. Modern vector databases are tuned to achieve a target mean recall while maximizing throughput [8, 15, 55, 63]. We find that when optimizing for the mean recall alone, recall can differ significantly across embedding space regions even when the target mean recall is met.

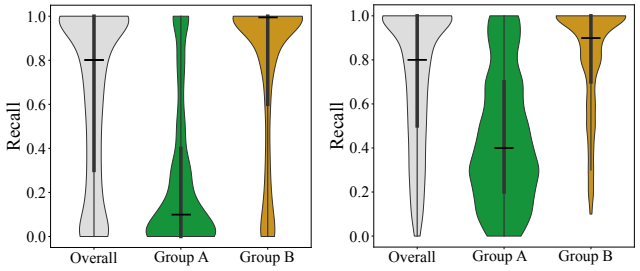
To characterize recall differences across queries, we organize stored vectors into *groups* based on their location in the embedding space using popular k-means clustering [2] during offline index construction. The k centroids represent distinct embedding space regions. We map each query to the group whose centroid is nearest to it using the same distance metric used by the underlying index. Such grouping helps us conveniently analyze whether queries that map to a certain embedding space region face significantly different recall compared to queries that map to another region.

Fig. 1 shows the recall distribution for two such groups (group A and group B). Experimental setup is the same as the one described in §1. Despite using identical search configurations and covering comparable portions of the dataset, group A achieves a mean recall of only 0.21, while group B achieves 0.99. This example shows that queries targeting different regions of the embedding space can experience significantly different recall.

Why identify and reduce recall differences? The recall differences illustrated above have consequences for both users and the vector search system. Queries in low-recall regions miss relevant results, even when the overall mean recall appears to satisfy the target recall. As noted in in § 1, recall is correlated with user satisfaction and can lead to user disengagement and/or negatively impact the end application that is using the vector search system.

An interesting observation is that these recall differences can align with semantic groups. For example, in Fig.1, post-hoc analysis shows that over 90% of the words in the low-recall group A correspond to Arabic tokens, while over 92% of the words in a high-recall group B are Latin-script tokens. We emphasize that this observation does not imply inherent superiority of one language over another. Rather, it highlights that ANN index behavior can interact with embedding geometry in ways that disproportionately affect specific semantic groups. When such group-level recall differences align with meaningful query groups, low recall is no longer an isolated issue affecting individual queries, but a systematic problem affecting entire classes of users.

Why do potential offline solutions fall short? A possible solution is to reduce recall differences across queries through offline solutions, such as, switching embedding models, using another index or rebuilding the index with more aggressive construction parameters. However, these solutions



(a) GloVe-200 with HNSW **(b)** GloVe-100 with IVFFlat

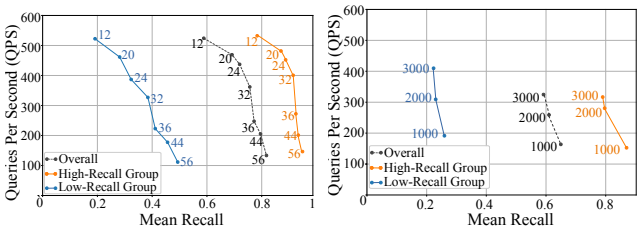
Figure 2. Recall distributions across all queries and across the low-recall and high-recall groups for two ANN index configurations: (a) GloVe-200 with HNSW and (b) GloVe-100 with IVFFlat. Even when the overall average recall differs across configurations, recall remains widely spread, with substantial gaps between low-recall and high-recall groups.

are insufficient. They are expensive and disruptive in production: they require recomputing embeddings and/or rebuilding the index, increasing computational cost and memory footprint, and complicating deployment in production environments [45]. Moreover, we find that recall differences can persist even after applying these solutions, and when they do help, it often comes at the cost of significant throughput degradation. We show this through the case studies below.

Switching embedding model. One might try to address recall differences by switching to a different embedding model. To test this, we replace the GloVe-100 embedding model with GloVe-200 [50], a different model that represents each word as a 200-dimensional vector instead of 100 dimensions, on the same Twitter dataset [49] used in Fig.1.

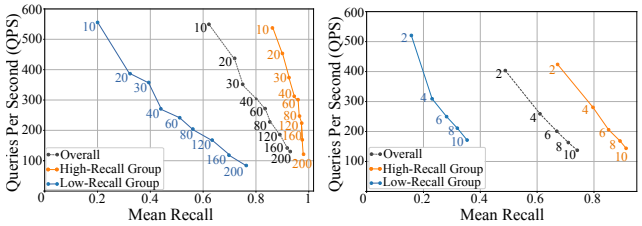
Fig. 2a shows recall distribution for this higher-dimensional embedding model. Similar to GloVe-100, the overall recall distribution is wide, with significant variation across queries. When we partition the embedding space into groups and examine recall at the group granularity, we see that recall differences persist across groups. For example, two groups of similar size (3% of the dataset each) exhibit vastly different recall: group B achieves mean 0.81 (25th-75th percentile: 0.6-1.0) vs. group A's mean of 0.26 (25th-75th percentile: 0-0.4). We do not claim that all embedding models exhibit identical recall behavior. However, this case study shows that recall differences across groups are not artifacts of a specific embedding model and they can persist across commonly used embedding representations, even if their magnitude varies.

Switching ANN index types. One might hypothesize that recall differences are specific to a particular ANN index type. To test this, we replace the graph-based HNSW index [51] with IVFFlat [60], a partition-based index, on the same Twitter dataset with GloVe-100 embeddings as in Fig. 1. Fig. 2b shows the resulting recall distribution. We observe that, although IVFFlat changes the absolute recall values, the overall distribution remains wide. When we examine recall at the



(a) HNSW: Varying `M` **(b)** IVFFlat: Varying `nlist`

Figure 3. QPS–recall trade-offs for the low-recall and high-recall groups and the overall distribution as index construction parameters are varied: (a) HNSW with fixed `ef_construction`=200 and `ef_search`=20 while varying `M`, and (b) IVFFlat with fixed `probes`=4 while varying `nlist`. Although tuning these parameters improves recall for the low-recall group, it degrades QPS; also, the same parameter changes affect recall improvement differently across groups.



(a) HNSW: Varying `ef_search` **(b)** IVFFlat: Varying `probes`

Figure 4. QPS–recall trade-offs for the lowest- and highest-performing groups and the overall distribution as HNSW's `ef_search` (a) and IVFFlat's `probes` (b) are varied. Increasing search effort improves recall for low-recall groups but also incurs unnecessary work for high-recall groups, leading to degrading throughput; also, identical increases in search effort yield different recall improvements across groups.

group granularity, these differences persist: groups of comparable size exhibit substantially different recall behavior, e.g., group A: mean 0.44, 25th-75th percentile: 0.2-0.7; group B: mean 0.82, 25th-75th percentile: 0.7-1.0. This demonstrates that recall differences are not specific to an ANN index and can persist across different index types.

Tuning index construction parameters. We examine whether tuning index construction parameters can mitigate recall differences. To test this, we use the Twitter dataset [49], GloVe-100 embedding model, and grouping as in Fig. 1.

For HNSW, we vary `M` (the number of connections per vector), which controls graph density. Higher `M` improves recall at the cost of increased memory and reduced throughput. Fig. 3a plots QPS vs. mean recall for two example groups, one low-recall and one high-recall, alongside the overall distribution. Although higher `M` improves recall both overall and within individual groups, significant recall differences persist. Moreover, groups do not benefit equally from increasing `M`: at `M`=12, the low-recall group achieves 0.19 vs. 0.78 for the high-recall group; at `M`=56, these differences persist at 0.50 and 0.95, respectively. Increasing `ef_construction` (the search effort during index building) follows a similar trend: absolute recall improves, but recall differences remain.

We observe the same pattern for IVFFlat (Fig. 3b). We vary `nlist` (the number of partitions) while keeping `nprobes` (partitions searched per query) fixed . Increasing `nlist` creates smaller partitions; however, with constant `nprobes`, overall recall decreases as nearest neighbors increasingly fall outside the searched partitions. Recall differences across groups persist nonetheless: at `nlist`=1000, low-recall group achieves mean recall of 0.27 vs. 0.95 for the high-recall group; at `nlist`=3000, mean recall is 0.22 vs. 0.89, respectively.

This illustrates that tuning index construction parameters does not eliminate recall differences and even when such tuning reduces recall differences, it often comes at the cost of significantly reduced throughput.

Why do typical run-time solutions fall short? At run-time, operators can tune a *search effort* parameter (e.g., `ef_search` for HNSW or `nprobes` for IVFFlat), but this is typically configured for the entire index. Increasing search effort helps low-recall groups, but it also increases work for groups that already achieve satisfactory recall, leading to unnecessary computation and reduced net throughput.

Fig. 4a illustrates this trade-off for HNSW: as `ef_search` increases, recall for low-recall groups improves, but net throughput degrades due to extra work on high-recall groups. Moreover, increasing search effort can benefit each group differently. At `ef_search`=20, the low-recall group achieves recall of 0.32 vs. 0.90 for the high-recall group; at `ef_search`=160, these become 0.70 and 0.97, respectively.

Fig. 4b shows a similar pattern for IVFFlat when we increase `nprobes`. At `nprobes`=2, the low-recall group achieves recall of 0.18 vs. 0.67 for the high-recall group; at `nprobes`=10, these become 0.38 and 0.97, respectively.

This underscores that global run-time tuning cannot eliminate recall differences and improvements for low-recall groups result in unnecessary computation for groups that already perform well, reducing overall throughput.

Takeaway. Both ANN index construction and run-time tuning fall short in reducing recall differences across queries without significantly hurting overall throughput. This motivates a finer-grained approach: selectively adjusting search effort per group based on recall profiles by concentrating more effort for low-recall groups and avoiding unnecessary work for groups that already meet the target recall. We design *RCheck* to enable this finer-grained approach.

4 RCheck

Building on the observations in §3, we introduce *RCheck*, a light-weight run-time adaptation system that detects groups

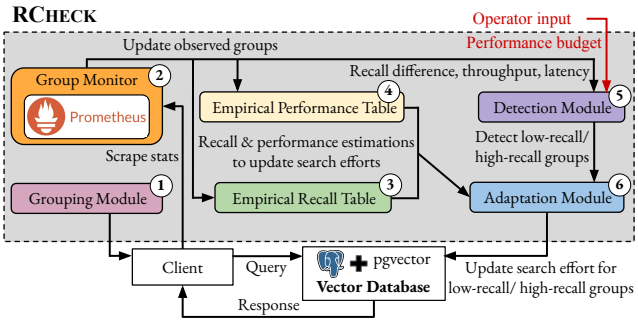


Figure 5. *RCheck* system design overview.

whose recall diverges from the target recall and applies fine-grained, group-specific search adaptations to reduce these differences. Before detailing *RCheck* 's modules, we describe design goals that shape *RCheck*'s architecture and workflow.

4.1 RCheck Design Goals

RCheck's goal is to balance recall across queried regions of the embedding space at run-time by increasing search effort for low-recall groups while reducing unnecessary work for groups whose recall already exceed the target. To achieve this goal, we design *RCheck* around three principles:

Run-time, non-intrusive operation. *RCheck* operates entirely at run-time without modifying embeddings, rebuilding the ANN index, or changing the database architecture. This ensures *RCheck* is easy to deploy in production and avoids expensive offline reconfiguration.

Fine-grained search effort adaptation. Common index configuration practices to address recall differences across queries fall short. They either cause queries that map to some regions of the embedding space to face low recall or lead to unnecessary work for queries that map regions that achieve high recall even with low search effort, degrading overall throughput. *RCheck* avoids this by adjusting search effort per group at run-time, guided by each group's recall profile.

Balancing the recall vs. throughput tradeoff. Improving recall can hurt throughput. To balance this, *RCheck* lets operators define performance constraints (e.g., throughput), then redistributes search effort across groups: reducing effort for groups exceeding the target recall and increasing it for those falling short, balancing recall while staying within the operator-defined performance budget.

4.2 RCheck High-level Overview

RCheck operates as a feedback-driven control-loop, as shown in Fig. 5. At run-time, it continuously monitors recall and performance metrics, including throughput and tail latency, for each group, maintaining light-weight empirical tables that summarize how recall and performance respond to changes in search effort. Periodically, *RCheck* uses these tables to adjust per-group search effort, correcting recall deviations from the target while staying within the throughput and tail latency budget defined by the operator.

To enable fine-grained control, *RCheck* organizes queries into *groups*, which serve as the units of monitoring and adaptation. Per-query adaptation would be ideal but is computationally expensive: tracking individual query behavior and computing ground truth for every query is impractical at scale. Groups provide a practical middle ground by grouping queries that map to the same region of the embedding space (§3). Groups are defined at index construction time using k-means clustering [44] and are purely logical and impose no changes to the underlying index structure.

We now describe *RCheck*'s run-time adaptation's stages.

Group query assignment. At run-time, *RCheck* assigns each incoming query to a group via nearest-centroid lookup.

Per-group monitoring. *RCheck* operates in fixed-length monitoring intervals. During each monitoring interval, *RCheck* samples queries from every group and performs exact searches in parallel to obtain true neighbors (i.e., ground truth). Comparing these with the ANNS results estimates per-group recall. *RCheck* also records per-group throughput and tail latency observed during the interval.

Per-group adaptation. At the end of each monitoring interval, *RCheck* adjusts search effort as follows:

(i) Empirical table update. Using per-group statistics collected during the monitoring interval, *RCheck* updates two light-weight summaries for each group: a *recall table* recording how recall varies with search effort, and a *performance table* recording the corresponding throughput and latency. These tables are updated incrementally at runtime.

(ii) Recall difference detection. For each group, *RCheck* computes the difference between the recall observed in the current monitoring interval and the target recall. Groups are then ranked by the recall difference magnitude, so those furthest from the target are considered first for adaptation.

(iii) Search effort adaptation. RCheck processes groups in the ranked order, determining whether to increase, decrease, or maintain each group's search effort. For groups below target recall, *RCheck* consults the recall table to identify the smallest search effort increase estimated to shift recall towards the target. For groups well above target, *RCheck* considers reducing search effort only if the recall table indicates recall will remain above target and the performance table estimates meaningful throughput savings.

RCheck evaluates each proposed change against the operator-defined performance budget, accounting for all previously accepted adaptations. Adaptations that fit within the budget are accepted; others are skipped. By processing groups in ranked order, *RCheck* ensures that those with the largest recall deficits receive more search effort first, while those with the largest recall surplus contribute throughput savings.

(iv) Applying the search effort adaptation. RCheck applies the accepted search effort adaptation to the ANN index, which takes effect immediately. The next monitoring interval

then begins, during which, *RCheck* observes how the updated search efforts affect recall and throughput. This feedback informs the next round of search effort adaptations.

4.3 RCheck Modules

We now describe the modules that implement *RCheck*'s workflow (§4.2), as illustrated in Fig. 5.

① **Grouping module**: It creates groups at offline index construction time and maps incoming queries to them at runtime. By default, *RCheck* partitions the embedding space using k-means clustering [44] over indexed vectors, producing k centroids that represent distinct regions of the embedding space. Importantly, the grouping is purely logical and does not modify the underlying ANN index structure or internal database data structures.

At run-time, each incoming query is mapped to a group using nearest-centroid lookup: the query embedding is compared against the k centroids, and the query is mapped to the group with the smallest distance, enabling fine-grained, per-group monitoring and search effort adaptation.

② **Group monitor**: It continuously collects per-group statistics during each monitoring interval. For each group, it samples a small number of queries and executes exact searches in parallel to obtain the true nearest neighbors (i.e., ground-truth). It then calculates the recall by comparing these ground-truth results with the ANNS results returned along the normal query serving path. The group monitor also records per-group density (i.e., query count), throughput, and p99 tail latency observed during the monitoring interval. All monitoring and exact-search executions run asynchronously and do not block query processing.

③,④ **Empirical recall and performance tables.** For each group, *RCheck* maintains two empirical tables: (i) a *recall table* that records observed recall values under different search effort settings and (ii) a *performance table* that records the corresponding throughput and p99 latency. Each table stores <search effort, metric> pairs collected during monitoring intervals, capturing how changes in search effort affect recall and performance at group granularity.

RCheck uses these tables to estimate the direction and magnitude of change in recall or throughput as search effort is adjusted, enabling incremental, gradient-like updates at run-time without fitting an explicit model. The tables become more accurate as observations accumulate. When historical per-group statistics are available from a previous execution, *RCheck* can optionally load them to initialize the tables and reduce cold-start overhead.

⑤ **Detection module**: At the end of each monitoring interval, this module compares the measured recall of each group with the target recall. It computes the recall differences and ranks groups by difference magnitude, ensuring those furthest from the target are considered first for adaptation. Groups below the target are marked for increased search effort; those above target are candidates for reduction.

Algorithm 1: *RCheck* Intervention

Input: ranked groups $\mathcal{G}$; recall table T_r; performance table T_p; target recall r^*; throughput budget B_t; latency threshold L_{max}

1 $\mathcal{A} \leftarrow \emptyset$; // Accepted adaptations
2 $b^{\text{rem}} \leftarrow B_t$; // Remaining throughput budget
3 **foreach** *group* $g \in \mathcal{G}$ *in order of* $|r_g - r^*|$ **do**
4 **if** $|r_g - r^*| < \epsilon$ **then**
5 **continue**; // Within tolerance
6 $(\Delta e, \Delta b) \leftarrow \textsc{GetEffort}(g, r^*, T_r, T_p, b^{\text{rem}}, L_{max})$;
7 **if** $\Delta e \neq 0$ **then**
8 $\mathcal{A} \leftarrow \mathcal{A} \cup \{(g, \Delta e)\}$;
9 $b^{\text{rem}} \leftarrow b^{\text{rem}} - \Delta b$;
10 $\textsc{ApplyAdaptation}(\mathcal{A})$;

11 **Function** $\textsc{GetEffort}(g, r^*, T_r, T_p, b^{rem}, L_{max})$**:**
12 $\Delta e_r \leftarrow \textsc{RecallDrivenAdaptation}(g, r^*, T_r)$;
13 **if** $r_g < r^*$ **then**
  // Low-recall group: Increase effort
14 $\Delta e_b \leftarrow \textsc{BudgetDrivenLimit}(g, T_p, b^{\text{rem}})$;
15 $\Delta e_l \leftarrow \textsc{LatencyDrivenLimit}(g, T_p, L_{max})$;
16 $\Delta e \leftarrow \min(\Delta e_r, \Delta e_b, \Delta e_l)$;
17 **if** $\Delta e \leq 0$ **then**
18 **return** $(0, 0)$; // Violates budget
19 **else**
  // High-recall group: Decrease effort
20 $\Delta e \leftarrow \Delta e_r$
21 **if** $\Delta e < \lambda$ **then**
22 **return** $(0, 0)$; // Violates target
23 **if** $\textsc{ThroughputGain}(g, \Delta e, T_p) < \gamma$ **then**
24 **return** $(0, 0)$; // Gain too small
25 $\Delta b \leftarrow \textsc{EstimateCost}(g, \Delta e, T_p)$;
26 **return** $(\Delta e, \Delta b)$;

⑥ **Adaptation module**: It balances recall across the queried portion of the embedding space by adjusting per-group search effort to bring each group closer to the target recall, subject to operator-defined performance budgets. Algorithm 1 summarizes this process.

The module processes groups in ranked order, prioritizing those with the largest recall difference from the target r^* (line 3). Groups whose recall falls within tolerance ϵ of the target require no adaptation and are skipped (lines 4–5).

For each remaining group, the GetEffort function determines the appropriate search effort adaptation Δe and its associated estimated throughput cost Δb (line 6).

The GetEffort function first computes the recall-driven adaptation Δe_r (line 12) using the recall table T_r. T_r records, for each group, the observed recall at previously used effort

levels. From these observations, the function estimates the recall slope for the group: the expected change in recall per unit change in effort. The recall-driven adaptation is then computed as the effort change needed to close the gap between the current recall r_g and the target r^*. For low-recall groups, this adaptation is positive (increase effort); for high-recall groups, it is negative (decrease effort). The subsequent logic determines whether this adaptation is acceptable given the performance constraints.

For low-recall groups, the function computes two additional limits (lines 14–15): the budget-driven limit Δe_b, representing the maximum effort increase that the remaining throughput budget can accommodate, and the latency-driven limit Δe_l, representing the maximum increase that keeps the group's mean latency below the threshold L_{max}. The function selects the minimum of these three candidates, ensuring the adaptation improves recall as much as possible without violating either limit (line 16). If no positive adaptation is feasible, the group is skipped (lines 17–18).

For high-recall groups, the function GetEffort considers reducing effort to avoid unnecessary work and reclaim throughput budget. It uses the recall-driven adaptation directly, which is negative in this case (line 20). However, not all search effort reductions are worthwhile: if the reduction magnitude is smaller than the threshold λ, the adaptation is skipped (lines 21–22). This prevents reductions that would bring recall too close to the target, preserving a safety margin against estimation error. Moreover, if the estimated throughput gain falls below the threshold γ, the reduction is skipped (lines 23–24). This prevents unnecessary adaptations that yield negligible throughput benefit.

For accepted adaptations, the function GetEffort estimates the throughput cost Δb from the performance table T_p (line 25). The cost is positive for search effort increases (consuming budget) and negative for effort decreases (reclaiming budget). Accepted adaptations are collected, and the remaining budget is updated accordingly (lines 7–9). After processing all groups, the adaptations are applied atomically (line 10). Reclaimed budget from high-recall groups becomes available for subsequent low-recall groups, enabling the adaptation module to redistribute search effort toward regions of the embedding space that need it most.

4.4 Implementation

We implement *RCheck* in ~2500 lines of Python code on top of the ANN Benchmarking framework [15]. Our implementation requires no modification to the underlying vector database or ANN index. *RCheck* has four components that realize the modules described in §4.3: a query router to implement grouping module, a metrics collector and recall estimator to implement the group monitor, and a controller to implement the detection module and adaptation module.

Query router. It implements the grouping module and enforces per-group search effort. The grouping module maps each incoming query to its corresponding group by computing the nearest centroid, as described in § 4.3. After determining the group, the router retrieves the current search effort for that group from a thread-safe dictionary maintained by the *RCheck* controller. It then injects the search effort value by issuing a PostgreSQL `SET LOCAL` command immediately before the vector search. This approach requires no modifications to pgvector and only access to the standard PostgreSQL parameter interface. All dictionary lookups and updates use atomic compare-and-swap operations to ensure queries never observe inconsistent state.

Metrics collector. It implements the group monitor and collects the statistics needed for the empirical tables. We deploy a Prometheus [7] instance alongside pgvector, coordinating both through Docker Compose. At run-time, the metrics collector records per-group statistics: query count (n_g), throughput, latency, and current search effort (e_g). These metrics are exposed via HTTP endpoints and scraped by Prometheus at a configurable *monitoring interval*. Shorter intervals enable faster adaptation to workload shifts but increase estimate variance and computation cost; longer intervals yield more stable measurements at the cost of slower response to recall differences from the target. Because Prometheus pulls metrics asynchronously, the monitoring interval has no impact on query latency.

Recall estimator. It implements the recall estimation functionality in the group monitor. We estimate per-group recall by sampling 2% of queries per group during each monitoring interval. For each sampled query, a background thread executes an exact nearest-neighbor search (brute-force scan) and compares the result against the ANN result to compute recall. These exact searches run in a dedicated thread pool and do not block the primary query path. At the end of each monitoring interval, the estimator computes the per-group recall estimate (r_g) as the mean recall across all samples and exports it to Prometheus alongside the other metrics.

Controller. It implements the detection module and adaptation module and maintains the empirical recall and performance tables (T_r and T_p). It runs as a dedicated process and coordinates the feedback loop at the end of each monitoring interval through three steps:

Step 1: Collect statistics. The controller queries Prometheus to retrieve per-group metrics: query count, recall estimate, latency, and current search effort.

Step 2: Update tables and detect recall differences. It updates the empirical recall table T_r and performance table T_p with the new observations. From these observations, the controller estimates how recall and latency change with search effort for each group using linear regression. The detection module then identifies groups where $|r_g - r^*| > \epsilon$, flagging them for intervention.

Step 3: Compute and apply search effort adaptations. For each flagged group, the adaptation module computes new search effort values using Algorithm 1. Updated values are

pushed to the query router via a single gRPC [34] call, which atomically refreshes the per-group effort dictionary.

Our implementation decouples monitoring (Prometheus pulls metrics from the query router) from control (the controller pushes search effort adaptations), allowing each component to operate independently without blocking queries or introducing synchronization overhead.

Parameters. We configure *RCheck* with $k \in \{10, 50\}$ groups depending on dataset size: $k = 10$ for smaller datasets (FairFace, CIFAR-100, DBPedia) and $k = 50$ for the larger dataset (GloVe-100). We use a monitoring interval of 15 seconds. Search effort values follow ANN benchmarking defaults [15]: `ef_search` $\in [10, 500]$ for HNSW and `nprobe` $\in [1, 50]$ for IVF. Any discrete value within this range is valid. The intervention thresholds, recall tolerance $\epsilon = 0.02$, safety margin $\lambda = 0.05$, and minimum throughput gain $\gamma = 5\%$, are set conservatively to reduce oscillation and filter out negligible adaptations. We evaluate sensitivity to k and the monitoring interval in §6.

5 Experimental Methodology

Experimental setup. We evaluate *RCheck* using the widely-used pgvector[6], a vector similarity search extension for PostgreSQL16, as the underlying vector database [3–5, 15]. *RCheck* is integrated directly into the ANN Benchmarking framework [15], operating at the client layer without modifications to the database itself.

Our experiments run on a c6420 CloudLab node [26] with dual 16-core Intel Xeon CPUs (32 cores total) and 365GB of RAM. We allocate resources across containerized services deployed via Docker Compose. The pgvector service receives 32GB of memory and 8 CPU cores. *RCheck* is deployed via Docker Compose alongside pgvector. Prometheus monitoring runs in a separate container with 1 CPU core and 256MB of memory; empirically, it uses less than 100MB of memory and <1% CPU during typical 15-second monitoring intervals, so our allocation provides conservative headroom.

Datasets. We evaluate *RCheck* on datasets spanning text, face recognition, vision, and knowledge-graph domains [14, 39, 41, 50]. For each dataset, we use a 90/10 train-test split: the training set (90%) is indexed into the vector database, while the test set (10%) composes the query workload.

(i) *GloVe.* This is a text dataset made of 1.18M pre-trained word vectors (50–200 dimensions) [50]. These vectors were learned from a very large collection of public Twitter posts written in different languages [49].

(ii) *FairFace.* It consists of 108,000 face images [39]; we embed all images using the state-of-the-art InsightFace model [30], producing 512-dimensional vector embeddings.

(iii) *CIFAR-100.* It contains 60,000 images evenly distributed across 100 semantic classes [41]; we generate 512-dimensional embeddings using widely-used CLIP ViT-B/32 [53].

(iv) *DBPedia.* It is derived from Wikipedia and represents structured knowledge-graph entities such as people, locations, and artworks [14]. To avoid over-representation of highly popular entities, we construct a balanced subset by sampling page titles evenly across categories and alphabetic prefixes. We embed each page title into a 384-dimensional vector using the popular `all-MiniLM-L6-v2` sentence embedding model [1], producing 53,000 vectors.

Compared configurations. We evaluate *RCheck* against pgvector and an ideal adaptation system.

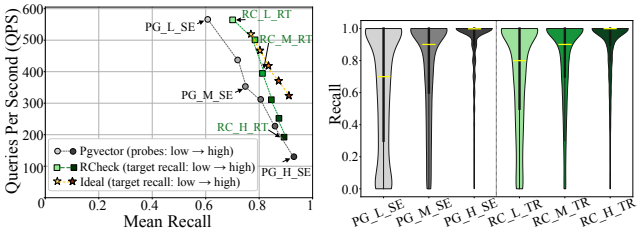
(1) pgvector. We use pgvector as our baseline because it is one of the widely deployed open-source vector database extensions [3–5, 15]. Pgvector supports two index types: HNSW [51] and IVFFlat [60]. We evaluate both to show that *RCheck* generalizes across these index structures. For each index, we sweep the global search effort parameter applied uniformly to all queries: `ef_search` for HNSW and `probes` for IVFFlat. This sweep produces a curve characterizing the throughput-recall trade-off achievable with global tuning. We highlight three representative operating points used throughout our evaluation: PG_L_SE (low search effort, high throughput, lower recall); PG_M_SE (medium search effort, balanced throughput and recall); and PG_H_SE (high search effort, lower throughput, higher recall).

(2) RCheck. We evaluate *RCheck* by sweeping the *target recall* parameter. This sweep produces a curve characterizing the throughput-recall trade-off achievable with group-level adaptation. We examine three representative operating points used throughout our evaluation: RC_L_RT (low target recall, high throughput, lower recall); RC_M_RT (medium target recall, balanced throughput and recall); and RC_H_RT (high target recall, lower throughput, higher recall).

(3) Ideal adaptation system. We include an ideal adaptation system that represents the best achievable throughput-recall trade-off under per-query adaptation. Given a target recall, the ideal adaptation system assigns each query its individually optimal search effort (i.e., the minimum value needed to achieve the target recall for that specific query). This is unattainable in practice since it requires knowing each query's true nearest neighbors at run-time. However, it quantifies the maximum gains achievable through perfect per-query adaptation and serves as an upper bound for *RCheck*. We sweep the target recall to produce a curve comparable to pgvector and *RCheck*.

6 Evaluation

We answer five questions. **(Q1)** How does *RCheck* reshape the throughput-recall trade-off compared to pgvector and ideal adaptation system, including effects on tail latency? **(Q2)** How does the monitoring interval affect *RCheck*'s effectiveness? **(Q3)** How does the number of clusters impact *RCheck*'s behavior? **(Q4)** How robust is *RCheck* to changes in ANN search parameters? **(Q5)** What is *RCheck*'s overhead?



(a) Mean recall vs. QPS. **(b)** Recall distributions.

Figure 6. (a) QPS-recall trade-off comparison: *RCheck* vs. pgvector vs. the ideal on GloVe-100 with HNSW. (b) Recall distributions across *RCheck* and pgvector for selected points.

6.1 Evaluating *RCheck*'s Effectiveness

6.1.1 GloVe-100. We run it on HNSW with $k = 50$ groups.

QPS-recall tradeoff. Fig. 6a shows QPS vs. mean recall for GloVe-100 using an HNSW index. Each curve represents the full range of achievable QPS vs. mean recall trade-offs for pgvector, *RCheck*, and ideal adaptation system, obtained by sweeping corresponding tuning parameters, i.e., `ef_search` for pgvector; target recall for *RCheck* and ideal adaptation.

The *RCheck* curve lies strictly above and to the right of the pgvector curve, achieving higher recall at any given throughput and higher throughput at any given recall. For example, at target recall 0.85, *RCheck* achieves 40.91% higher throughput than pgvector. At 400 QPS, *RCheck* improves recall from 0.73 to 0.81, a gain of 10.96%. These gains move *RCheck* closer to the ideal curve. The ideal adaptation system achieves 90.48% higher throughput than pgvector at 0.85 recall. *RCheck* closes 45% of the gap to ideal adaptation system. The remaining gap exists because *RCheck* adapts at the group level rather than per query. Per-query adaptation would require tracking the impact of search effort on each query individually, necessitating ground truth computation for every query, a high cost. Nonetheless, *RCheck* achieves 40.91% higher throughput than pgvector, which does not adapt to reduce recall differences.

In deployment, operators do not need to sweep the target recall parameter. Instead, they specify their requirements: a target recall and a performance constraint (e.g., throughput). *RCheck* then automatically finds a search effort allocation across groups that satisfies both. However, not all requirements are achievable. The *RCheck* trade-off curve in Fig. 6a determines what is achievable. If the requirements are infeasible, e.g., demanding 0.95 recall at 500 QPS when the curve shows 0.95 recall is only achievable at 200 QPS, *RCheck* reports failure, and the operator must relax the recall target, the performance constraint, or both.

Recall distribution . Beyond mean recall, we examine how *RCheck* reshapes the per-query recall distribution. Fig. 6b compares pgvector and *RCheck* at three pairs of operating points: (PG_L_SE vs. RC_L_RT) and (PG_M_SE vs. RC_M_RT) achieve similar throughput, while (PG_H_SE vs. RC_H_RT) achieves similar mean recall, as described in §5. To enable fair comparison across configurations that may use different internal target recalls, we evaluate all distributions against a fixed reference target of 0.85 (Ref_Tgt).

We characterize each recall distribution using three metrics: (1) the fraction of queries meeting or exceeding Ref_Tgt, which measures how consistently the system delivers acceptable search quality [62]; (2) the mean absolute error (MAE) relative to Ref_Tgt; and (3) the expected shortfall below Ref_Tgt, which is borrowed from risk analysis [11, 56], measures the average recall deficit among queries that fail to meet Ref_Tgt. Together, these metrics quantify how often queries fail to meet Ref_Tgt (fraction below Ref_Tgt), how much queries typically deviate from the target (MAE), and how severe the worst-case failures are (shortfall).

The pgvector at PG_L_SE produces a wide distribution with 25th/75th percentiles of 0.3/1.0, spanning 0.7 recall points. Only 38.7% of queries meet the Ref_Tgt, with MAE of 0.33 and shortfall of 0.29, indicating that queries missing the Ref_Tgt fall short by an average of 0.29 recall points. *RCheck* at matched throughput (RC_L_RT) tightens the 25th/75th percentiles to 0.5/1.0, reducing the spread to 0.5 recall points. The fraction of queries meeting Ref_Tgt increases to 46.7%, MAE improves to 0.26, and shortfall decreases to 0.20. This 31% reduction in shortfall means that when queries do fall short of the Ref_Tgt, they are closer to meeting it.

Moving to PG_M_SE, with pgvector, 55% of queries meet Ref_Tgt, with MAE of 0.35, shortfall of 0.17, and 25th/75th percentiles of 0.6/1.0. *RCheck* at comparable throughput (RC_M_RT) improves all three metrics: 62.6% meet Ref_Tgt, MAE decreases to 0.18, shortfall drops to 0.11 and the distribution tightens the 25th/75th percentiles to 0.7/1.0.

We compare PG_H_SE and RC_H_RT, which achieve similar mean recall, to evaluate how each distributes recall across queries. With PG_H_SE, 81.8% of queries meet Ref_Tgt (MAE: 0.15, shortfall: 0.04). With RC_H_RT at slightly lower mean recall, 78.3% of queries meet Ref_Tgt (MAE: 0.16, shortfall: 0.06). While RC_H_RT's distributional metrics are marginally worse, it achieves 81.11% higher throughput. This shows that even at high recall targets, *RCheck* delivers comparable search quality with substantially higher throughput.

Tail latency. For RC_L_RT vs. PG_L_SE, *RCheck* faces slightly higher tail (p99) latency (3.11 ms vs. 2.23 ms) as it allocates additional search effort to low-recall groups, while PG_L_SE applies minimal effort uniformly. At higher recall targets, *RCheck* significantly reduces tail latency. Compared to PG_M_SE, RC_M_RT reduces p99 latency from 17.28 ms to 6.61 ms; compared to PG_H_SE, RC_H_RT reduces p99 from 25.88 ms to 10.95 ms. Pgvector applies high search effort to all queries; *RCheck* avoids unnecessary work on high-recall queries, reducing tail latency.

6.1.2 FairFace. We run it on HNSW with $k = 10$ groups.

QPS-recall tradeoff. Fig. 7a shows the QPS-recall tradeoff, where *RCheck* bridges the gap between pgvector and the ideal adaptation system. At 0.85 recall, *RCheck* achieves 25.63% higher throughput than pgvector, closing 69.4% of the gap to ideal. At 187 QPS, *RCheck* improves mean recall from 0.56 to 0.73, an improvement of 30.36%.

Recall distribution . Fig. 7b shows the recall distributions at three operating points. The pgvector at PG_L_SE achieves 24.3% of queries meeting Ref_Tgt, with MAE of 0.35 and shortfall of 0.32 (25th/75th percentiles: 0.3/0.8). *RCheck* at matched throughput (RC_L_RT) improves to 38.2% meeting target, MAE of 0.22, and shortfall of 0.17 (percentiles: 0.5/0.9). At PG_M_SE, the pgvector shows 45.1% meeting target (MAE: 0.2, shortfall: 0.15, percentiles: 0.6/1), while *RCheck* (RC_M_RT) improves to 59.7%, 0.15, and 0.08 (percentiles: 0.7/1). For PG_H_SE, the pgvector achieves 81.8% meeting target (MAE: 0.13, shortfall: 0.03), while *RCheck* (RC_H_RT) at comparable mean recall shows 80.3% meeting target, MAE of 0.14, and shortfall of 0.04, with 19.41% higher throughput.

Tail latency. Tail latencies follow a similar trend as GloVe-100: for example, RC_M_RT vs. PG_M_SE, *RCheck* reduces p99 from 24.13 ms to 7.29 ms.

6.1.3 CIFAR-100.

We run it on IVFFlat with $k = 10$ groups to evaluate *RCheck* on a different index type.

QPS-recall tradeoff. Fig. 8a shows that *RCheck*'s benefits extend beyond HNSW. At 0.8 recall, *RCheck* achieves 6.95% higher throughput than pgvector, closing 59.1% of the gap to ideal adaptation system. At 200 QPS, *RCheck* improves recall from 0.42 to 0.81, a 92.86% gain.

Recall distribution The recall distribution improvements follow the same pattern as before. At PG_L_SE, the pgvector achieves 10.66% meeting Ref_Tgt, with MAE of 0.44 and shortfall of 0.43 (percentiles: 0.2/0.6), while *RCheck* (RC_L_RT) improves to 57.91% meeting target, MAE of 0.17, and shortfall of 0.1 (percentiles: 0.7/1). At PG_M_SE, the pgvector shows 50.8% meeting target, with MAE of 0.2 and shortfall of 0.13 (percentiles: 0.6/1), while *RCheck* (RC_M_RT) achieves 71.8% meeting target, MAE of 0.15, and shortfall of 0.05 (percentiles: 0.8/1). For PG_H_SE, the pgvector reaches 97.1% meeting target, with MAE of 0.14 and shortfall of 0, while *RCheck* (RC_H_RT) shows 92.3% meeting target, MAE of 0.14, and shortfall of 0.01, with 16.89% higher throughput.

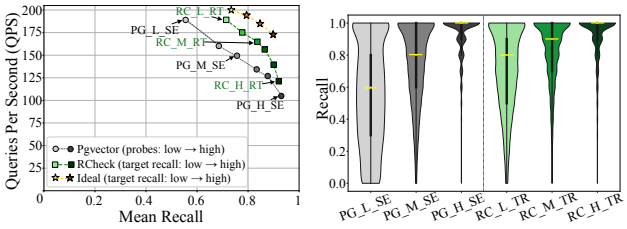
Tail latency. Tail latencies follow a similar trend as prior (e.g., 6.52 ms vs. 25.61 ms for RC_M_RT vs. PG_M_SE).

6.1.4 DBPedia.

We run it on IVFFlat with $k = 10$ groups.

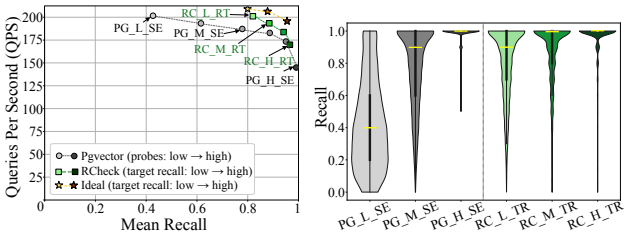
QPS-recall tradeoff. Fig. 9a shows that *RCheck* maintains its advantage across the operating range, achieving 23.6% higher throughput than pgvector at 0.85 target recall and covering 51.2% of the gap to ideal. At 225 QPS, *RCheck* improves mean recall from 0.71 to 0.84, a gain of 20%.

Recall distribution . The recall distribution analysis shown in Fig. 9b follows the prior pattern. At PG_L_SE, the pgvector achieves 31.73% meeting Ref_Tgt (MAE: 0.31,



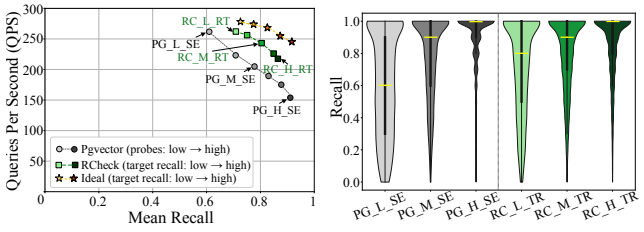
(a) Mean recall vs. QPS. **(b)** Recall distributions.

Figure 7. (a) QPS-recall trade-off comparison: *RCheck* vs. pgvector vs. the ideal on FairFace with HNSW. (b) Recall distributions across *RCheck* and pgvector for selected points.



(a) Mean recall vs. QPS. **(b)** Recall distributions.

Figure 8. (a) QPS-recall trade-off comparison: *RCheck* vs. pgvector vs. the ideal on CIFAR-100 with IVFFlat. (b) Recall distributions across *RCheck* and pgvector for selected points.



(a) Mean recall vs. QPS. **(b)** Recall distributions.

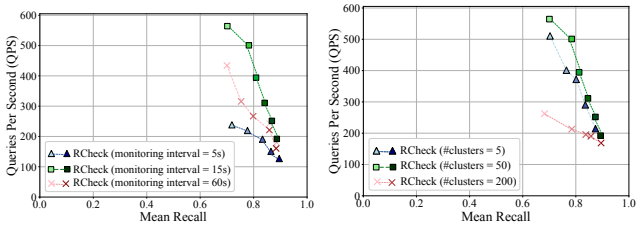
Figure 9. (a) QPS-recall trade-off comparison: *RCheck* vs. pgvector vs. the ideal on DBPedia with IVFFlat. (b) Recall distributions across *RCheck* and pgvector for selected points.

shortfall: 0.27, percentiles: 0.3/0.9), while *RCheck* (RC_L_RT) improves to 42.6%, 0.24, and 0.19 (percentiles: 0.5/1). At PG_H_SE, the pgvector reaches 77% (MAE: 0.14, shortfall: 0.03), while *RCheck* (RC_H_RT) shows 69.8%, 0.15, and 0.07, with 41.33% higher throughput.

Tail latency. Tail latency trends hold: RC_M_RT achieves 5.12 ms vs. 23.5 ms for PG_M_SE.

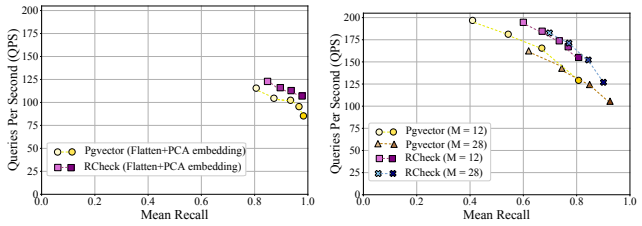
6.2 Evaluating the Monitoring Interval's Impact

We study how the monitoring interval affects *RCheck* by varying it between 5s, 15s, and 60s, shown in Fig. 10a. A short interval (5s) reacts quickly but relies on limited samples, leading to noisy recall estimates and frequent adaptations. At 0.85 target recall, this overhead reduces throughput to



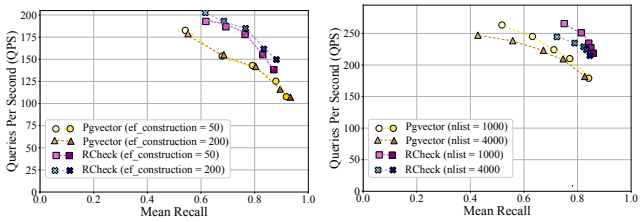
(a) Varying the monitoring interval **(b)** Varying the number of groups

Figure 10. QPS-recall trade-off on GloVe-100 with HNSW, showing *RCheck*'s sensitivity to (a) monitoring interval length and (b) number of groups.



(a) Varying embedding model. **(b)** Varying `M` in HNSW.

Figure 11. *RCheck* improves the QPS-recall trade-off on HNSW for the FairFace dataset (a) when changing the embedding model and (b) across different `M` values.



(a) Varying `ef_construction`. **(b)** Varying `nlist` in IVFFlat.

Figure 12. *RCheck* improves the QPS-recall trade-off on (a) HNSW for the FairFace dataset across different `ef_construction` values and (b) IVFFlat for the DBPedia dataset across different `nlist` values.

190 QPS compared to 310 QPS for 15s configuration, which is a 38.7% loss. At matched throughput of 240 QPS, mean recall drops to 0.72, underperforming 15s by 17.24%.

A long interval (60s) produces more stable estimates and reduces adaptation overhead, achieving 250 QPS at 0.85 target recall, which is better than 5s but still 19.35% below 15s. However, low-recall groups persist for up to 60s before adaptation, causing recall degradation: at 310 QPS, mean recall is 0.75, 11.76% below 15s.

The moderate interval (15s) balances stability and responsiveness. It achieves the best recall-throughput trade-off, operating closest to the ideal oracle across the entire range.

6.3 Group Granularity Impact on RCheck

We evaluate how the number of groups affects *RCheck* by varying group granularity while keeping the index and dataset fixed (Fig. 10b). With few groups (5), each group covers a large region of the embedding space, grouping queries with different recall behavior together. This limits *RCheck*'s ability to identify and correct localized recall problems: At the 0.85 target recall, this configuration achieves 280 QPS, which is a 10.71% throughput loss compared to 50 groups. At matched throughput (500 QPS), mean recall drops to 0.71, 10.13% below the 50-group configuration.

With many groups (200), each group becomes too small and receives too few queries for reliable recall estimation. This leads to frequent, noisy adaptations that increase overhead. At the 0.85 target recall, this configuration achieves only 200 QPS, which is 35.48% lower than 50 groups. At matched throughput (210 QPS), mean recall is 0.78, which is 11.36% worse than even the 5-group configuration.

Fig. 10b shows that 50 groups achieves the best balance. At this number of groups, groups are specific enough to isolate low-recall regions while receiving sufficient queries for stable recall estimation.

6.4 RCheck's Robustness to ANNS Parameters

We evaluate *RCheck*'s robustness to embedding model and index construction parameters (§3).

Embedding model variation. To assess sensitivity to embedding model choice, we repeat our experiments with alternative embeddings for FairFace: we flatten the original vectors and apply Principal Component Analysis (PCA) [9] to obtain 1024-dimensional embeddings. Fig. 11a shows that *RCheck* maintains its advantage: at 0.85 target recall, *RCheck* achieves 125 QPS compared to pgvector's 107 QPS (16.82% improvement), consistent with the behavior observed with the original embeddings.

HNSW construction parameters. For HNSW on FairFace, we evaluate robustness by varying key construction parameters from values used in §6.1. Our previous experiments used `M`=24; here we test with `M`=12 and `M`=28, shown in Fig. 11b. With `M`=12, *RCheck* achieves 153 QPS at 0.8 recall vs. pgvector's 126 QPS, which is a 21.4% improvement. With `M`=28, *RCheck* achieves 151 QPS at 0.85 recall vs. pgvector's 125 QPS, a 20.8% improvement.

Similarly, our previous FairFace experiment in §6.1 used `ef_construction`=100; here we test with `ef_construction`=50 and `ef_construction`=200, as shown in Fig. 12a. Across all parameter settings, *RCheck* consistently improves throughput by 20-40% at the target recall.

For IVFFlat (DBPedia), we similarly vary `nlist` by halving and doubling its default value as shown in 12b. *RCheck* again preserves its advantage across these settings, improving the trade-off between recall and throughput.

6.5 Evaluating *RCheck*'s Overhead

RCheck introduces the following overheads. Group creation is a one-time cost during setup; for $k \in \{5, 50\}$, it takes 260 seconds on average across datasets, adding ~30% overhead relative to index build time on average, and runs in parallel with index construction. Assigning each query to its group takes <0.1 ms, negligible compared to typical query latencies of 3–9 ms in our experiments. Recall estimation samples 2% of queries per group in each monitoring interval to estimate group-level recall, running in parallel with query processing and consuming <50% of CPU cycles. Runtime adaptation computes new search effort allocations in <14 ms, which is <0.1% of the 15-second monitoring interval and runs asynchronously without blocking queries.

7 Discussion

Run-time adaptation under performance constraints. We design *RCheck* to operate strictly at run-time without modifying the embedding model or the ANN index structure. This design choice avoids expensive index rebuilds and allows the system to react quickly to recall degradation under changing workloads. However, it also limits what *RCheck* can correct. *RCheck* assumes that the target recall is achievable within the available performance budget. If the target recall is unachievable because the index quality is too low or the performance budget is too tight, no amount of search effort tuning will suffice. *RCheck* detects this condition when groups persistently fall short of the target despite operating at maximum allowed search effort. In such cases, the system alerts the operator that the current recall target is unachievable within the performance budget, requiring offline intervention or revised constraints.

Search effort reallocation. A key insight behind *RCheck* is that groups exceeding the recall target can reduce their search effort to create computation headroom for groups falling short. This introduces coupling between groups: improving a group's recall may require reducing effort elsewhere. Our current design prioritizes groups by recall difference magnitude from the target, allocating more effort to those furthest below the target. This aligns with reducing maximum difference from the target recall. More expressive policies like protecting certain groups from effort reduction could be supported, but might need explicit operator input.

8 Related Work

Global parameter tuning. ANNS has been extensively studied, with widely deployed indexes including graph-based methods such as HNSW [45, 51] and DiskANN [35], and partition-based approaches such as IVFFlat [60], IVF [36], and FAISS [25]. These systems expose search effort parameters (e.g., `ef_search`) that trade recall for throughput. In practice, operators select global parameter values via benchmarking frameworks like ANN-Benchmarks [15] or vendor recommendations [6, 25], applying a single configuration to all queries. This approach assumes uniform recall behavior across queries and does not account for per-query or per-region variation in search quality. In contrast, *RCheck* operates at run-time and adapts search effort at a finer granularity based on observed recall.

Adaptive index structures Prior work has addressed workload heterogeneity by adapting ANN index structures. SPANN [20] identifies the closest clusters in memory and then reads only the necessary data blocks from the disk to find the nearest neighbors. This query-aware approach ensures the system only performs minimal disk reads, keeping search speeds fast. QUAKE [48] employs a multi-level partitioning scheme that reorganizes partitions in response to updates and changing access patterns, guided by a cost model predicting query latency. These approaches modify the index structure, requiring maintenance overhead and potential query disruption during reorganization. In contrast, *RCheck* operates at run-time by adopting search effort parameters without modifying the underlying index, enabling immediate adaptation.

Differences in search quality. Recent work has examined fairness in information retrieval, focusing on biases in ranked outputs [19, 58, 65] and in embedding representations [17, 18, 54]. For example, Singh et al. [58] propose exposure fairness for rankings, while Zehlike et al. [65] study group fairness in top-k results. These approaches primarily operate at the ranking or representation level. More recently, Wang et al. [62] also questioned the adequacy of mean recall as a system-level metric and introduced *robustness* as a metric that captures the consistency of recall across a workload.

RCheck is complementary to both lines of work. Rather than modifying embeddings or enforcing fairness at ranking time, *RCheck* focuses on disparities that arise within the ANNS process itself: even with fixed embeddings and ranking functions, ANN indexes can produce uneven recall across different regions of the embedding space. Moreover, while Wang et al. [62] focus on measuring recall variability, *RCheck* detects and reduces it at run-time by adapting search effort across query groups. We also adopt the proposed robustness as an evaluation metric (i.e., fraction below Ref_Tgt in §6) to quantify how effectively *RCheck* reduces recall variability.

9 Conclusion

We showed that the modern practice of optimizing for mean recall masks significant differences in recall across queries even when target recall is met, hurting user experience. To this end, we designed *RCheck*, a light-weight run-time system that identifies low-recall queries and reduces recall differences while achieving high throughput. *RCheck* tunes available search effort parameters, making it readily deployable. We evaluated *RCheck* using widely-used pgvector [6]. At the same throughput, *RCheck* reduces recall differences,

increasing mean recall by up to 93%, enabling 47% more queries to meet target recall compared to the state-of-the-art globally-tuned search configuration.

References

[1] [n. d.]. all-MiniLM-L6-v2.

[2] [n. d.]. KMeans.

[3] [n. d.]. Load vector embeddings up to 67x faster with pgvector and Amazon Aurora.

[4] [n. d.]. Load vector embeddings up to 67x faster with pgvector and Amazon Aurora.

[5] [n. d.]. Load vector embeddings up to 67x faster with pgvector and Amazon Aurora.

[6] [n. d.]. pgvector. https://github.com/pgvector/pgvector.

[7] [n. d.]. Prometheus. https://prometheus.io/.

[8] [n. d.]. Weaviate, ANN Benchmark. https://docs.weaviate.io/weaviate/benchmarks/ann.

[9] Hervé Abdi and Lynne J Williams. 2010. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics* 2, 4 (2010), 433–459.

[10] Mustafa Abualsaud et al. 2020. The effect of queries and search result quality on the rate of query abandonment in interactive information retrieval. In *Proceedings of the 2020 Conference on Human Information Interaction and Retrieval*.

[11] Carlo Acerbi and Dirk Tasche. 2002. Expected shortfall: a natural coherent alternative to value at risk. *Economic Notes* 31, 2 (2002), 379–388. doi:10.1111/1468-0300.00091

[12] Arkadeep Acharya, Brijraj Singh, and Naoyuki Onoe. 2023. Llm based generation of item-description for recommendation system. In *Proceedings of the 17th ACM conference on recommender systems*. 1204–1207.

[13] Azzah Al-Maskari and Mark Sanderson. 2010. A review of factors influencing user satisfaction in information retrieval. *Journal of the American Society for Information Science and Technology* (2010).

[14] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. Dbpedia: A nucleus for a web of open data. In *international semantic web conference*. Springer, 722–735.

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

[16] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[17] Tolga Bolukbasi, Kai-Wei Chang, James Y Zou, Venkatesh Saligrama, and Adam T Kalai. 2016. Man is to computer programmer as woman is to homemaker? debiasing word embeddings. *Advances in neural information processing systems* 29 (2016).

[18] Aylin Caliskan, Joanna J Bryson, and Arvind Narayanan. 2017. Semantics derived automatically from language corpora contain human-like biases. *Science* 356, 6334 (2017), 183–186.

[19] Carlos Castillo. 2019. Fairness and transparency in ranking. In *Acm sigir forum*, Vol. 52. ACM New York, NY, USA, 64–71.

[20] 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.

[21] Yannis Chronis, Helena Caminal, Yannis Papakonstantinou, Fatma Özcan, and Anastasia Ailamaki. 2025. Filtered vector search: State-of-the-art and research opportunities. *Proceedings of the VLDB Endowment* 18, 12 (2025), 5488–5492.

[22] Kenneth L Clarkson. 1994. An algorithm for approximate closest-point queries. In *Proceedings of the tenth annual symposium on Computational geometry*. 160–164.

[23] Fernando Diaz, Bhaskar Mitra, Michael D Ekstrand, Asia J Biega, and Ben Carterette. 2020. Evaluating stochastic rankings with expected exposure. In *Proceedings of the 29th ACM international conference on information & knowledge management*. 275–284.

[24] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The faiss library. *arXiv preprint arXiv:2401.08281* (2024).

[25] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2025. The faiss library. *IEEE Transactions on Big Data* (2025).

[26] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 1–14.

[27] Michael D Ekstrand, Robin Burke, and Fernando Diaz. 2019. Fairness and discrimination in recommendation and retrieval. In *Proceedings of the 13th ACM conference on recommender systems*. 576–577.

[28] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun, Haofen Wang, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997* 2, 1 (2023).

[29] Ramanathan Guha, Rob McCool, and Eric Miller. 2003. Semantic search. In *Proceedings of the 12th international conference on World Wide Web*. 700–709.

[30] Jia Guo and Jiankang Deng. [n. d.]. InsightFace: 2D and 3D Face Analysis Project. https://github.com/deepinsight/insightface.

[31] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. 2011. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, Vol. 22. 1312.

[32] Sariel Har-Peled, Piotr Indyk, and Rajeev Motwani. 2012. Approximate nearest neighbor: Towards removing the curse of dimensionality. (2012).

[33] Dengrong Huang, Zizhong Wei, Aizhen Yue, Xuan Zhao, Zhaoliang Chen, Rui Li, Kai Jiang, Bingxin Chang, Qilai Zhang, Sijia Zhang, et al. 2023. Dsqa-llm: Domain-specific intelligent question answering based on large language model. In *International Conference on AI-generated Content*. Springer, 170–180.

[34] Kasun Indrasiri and Danesh Kuruppu. 2020. *gRPC: up and running: building cloud native applications with Go and Java for Docker and Kubernetes*. O'Reilly Media.

[35] 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).

[36] 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.

[37] Zhi Jing, Yongye Su, and Yikun Han. 2025. When large language models meet vector databases: A survey. In *2025 Conference on Artificial Intelligence x Multimedia (AIxMM)*. IEEE, 7–13.

[38] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[39] Kimmo Karkkainen and Jungseock Joo. 2021. Fairface: Face attribute dataset for balanced race, gender, and age for bias measurement and mitigation. In *Proceedings of the IEEE/CVF winter conference on applications of computer vision*. 1548–1558.

[40] Shubhra Kanti Karmaker Santu, Parikshit Sondhi, and ChengXiang Zhai. 2017. On application of learning to rank for e-commerce search.

In *Proceedings of the 40th International ACM SIGIR conference on Research and Development in Information Retrieval*.

[41] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[42] 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.

[43] Xiangci Li and Jessica Ouyang. 2024. How Does Knowledge Selection Help Retrieval Augmented Generation? *arXiv preprint arXiv:2410.13258* (2024).

[44] Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE transactions on information theory* 28, 2 (1982), 129–137.

[45] 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.

[46] Falk Maoro, Benjamin Vehmeyer, and Michaela Geierhos. 2023. Leveraging semantic search and llms for domain-adaptive information retrieval. In *International Conference on Information and Software Technologies*. Springer, 148–159.

[47] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[48] Jason Mohoney, Devesh Sarda, Mengze Tang, Shihabur Rahman Chowdhury, Anil Pacaci, Ihab F Ilyas, Theodoros Rekatsinas, and Shivaram Venkataraman. 2025. Quake: Adaptive indexing for vector search. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 153–169.

[49] Jeffrey Pennington et al. [n. d.]. GloVe: Global Vectors for Word Representation. https://nlp.stanford.edu/projects/glove/.

[50] Jeffrey Pennington et al. 2014. Glove: Global vectors for word representation. In *Empirical methods in natural language processing*.

[51] pinecone. [n. d.]. Hierarchical Navigable Small Worlds (HNSW). https://www.pinecone.io/learn/series/faiss/hnsw/.

[52] Pinecone. [n. d.]. Locality Sensitive Hashing (LSH). https://www.pinecone.io/learn/series/faiss/locality-sensitive-hashing/.

[53] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning transferable visual models from natural language supervision. In *International conference on machine learning*. PmLR, 8748–8763.

[54] Navid Rekabsaz, Simone Kopeinik, and Markus Schedl. 2021. Societal biases in retrieved contents: Measurement framework and adversarial mitigation of BERT rankers. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*.

[55] 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.

[56] R. Tyrrell Rockafellar and Stanislav Uryasev. 2000. Optimization of conditional value-at-risk. *Journal of Risk* 2, 3 (2000), 21–42. doi:10.21314/JOR.2000.038

[57] Shelly Schwartz, Oleg Vasilyev, and Randy Sawaya. 2025. How important is Recall for Measuring Retrieval Quality? *arXiv preprint arXiv:2512.20854* (2025).

[58] Ashudeep Singh and Thorsten Joachims. 2018. Fairness of exposure in rankings. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 2219–2228.

[59] Jiahao Tian, Zhenkai Wang, Jinman Zhao, and Zhicheng Ding. 2024. Mmrec: Llm based multi-modal recommender system. In *2024 19th International Workshop on Semantic and Social Media Adaptation & Personalization (SMAP)*. IEEE, 105–110.

[60] Timescale. [n. d.]. Nearest Neighbor Indexes: What Are IVFFlat Indexes in Pgvector and How Do They Work. https://www.timescale.com/blog/nearest-neighbor-indexes-what-are-ivfflat-indexes-in-pgvector-and-how-do-they-work.

[61] 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.

[62] Qi Wang, Jindong Li, Shiqi Wang, Qianli Xing, Runliang Niu, He Kong, Rui Li, Guodong Long, Yi Chang, and Chengqi Zhang. 2024. Towards next-generation llm-based recommender systems: A survey and beyond. *arXiv preprint arXiv:2410.19744* (2024).

[63] Zikai Wang, Qianxi Zhang, Baotong Lu, Qi Chen, and Cheng Tan. 2025. Towards Robustness: A Critique of Current Vector Database Assessments. *arXiv preprint arXiv:2507.00379* (2025).

[64] Liang Wu, Diane Hu, Liangjie Hong, and Huan Liu. 2018. Turning clicks into purchases: Revenue optimization for product search in e-commerce. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*.

[65] Meike Zehlike, Francesco Bonchi, Carlos Castillo, Sara Hajian, Mohamed Megahed, and Ricardo Baeza-Yates. 2017. Fa* ir: A fair top-k ranking algorithm. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. 1569–1578.

[66] Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. 2023. Toolqa: A dataset for llm question answering with external tools. *Advances in Neural Information Processing Systems* 36 (2023), 50117–50143.