YOU DON'T HAVE TO GO THE DISTANCE

Anonymous authors
Paper under double-blind review

ABSTRACT

Neural models use large external memories accessed through learned representations. The cost of inference depends on the length of search for each query. Average recall gives equal weight to all recovered neighbors, while their recovery requires significantly different amounts of computation. We propose a learned search policy, TraCANN, that takes advantage of this asymmetry to reallocate the computation under a shared recall target. TraCANN's training trajectories provide the costs and neighbor recovery statistics used to fit continuation actions jointly, and it selects an action based on one policy evaluation from each query's early search progress. By learning the relative costs of recovery, TraCANN can stop promising searches before exhausting their candidate sets. Compared to eleven baselines, TraCANN's throughput is the highest at all evaluated recall thresholds on four standard ANN benchmarks. Our analysis demonstrates cases when such reallocations reduce computation and bounds the difference between discrete and fractional continuation assignments.

1 INTRODUCTION

Using learned representations, we can query large collections of text and image data using vector similarity (Reimers & Gurevych, 2019; Radford et al., 2021). Dense retrievers retrieve passages for question answering (Karpukhin et al., 2020) and nearest neighbor memories augment language models with information beyond their parameters (Khandelwal et al., 2020; Lewis et al., 2020; Borgeaud et al., 2022). Retrieval brings vector search to model inference, so retrieval efficiency directly impacts the cost of external memory.

Approximate nearest neighbor (ANN) search reduces this cost by exploring a small part of the database. Proximity graphs guide the search towards points close to the query. Compressed scoring allows candidate comparisons to be inexpensive (Malkov & Yashunin, 2020; Aguerrebere et al., 2023; Gou et al., 2025). Other savings come from how far the search continues. Even a promising route can require many visits to improve the retrieved neighbors. The contribution of a promising route to the average recall has to be balanced against the cost of finding those neighbors.

Average recall can be used to learn how to allocate this computation. Recovering an additional neighbor when the number of neighbors requested is fixed improves average recall by the same amount for any query. The computational cost to recover this neighbor depends on the search required to find it. On training queries, the computation can be reduced by trading an expensive improvement on one query for a less expensive improvement on another query, thus recovering the same number of neighbors. This ability to trade neighbors recovered for different queries reveals the conditions under which it is acceptable to stop while promising neighbors remain to be recovered. Learning this tradeoff from search progress leads to a rule for deciding how far to continue a new query.

We introduce TraCANN, Trajectory-Calibrated Approximate Nearest Neighbor Search, which learns how much of a graph traversal to perform based on early progress. A first stage of search retrieves neighbors and gathers evidence about their distances, local geometry, and recent changes in the search results. A small decision tree uses the evidence to determine how much further to search, using a frontier width and a total visit budget. The frontier width allows alternative search paths, and the visit budget limits the number of distance evaluations used to explore them. Search is performed from the current state, so the search computation used to determine the decision also contributes to the answer.

The policy is trained by measuring costs and neighbor recoveries along continuations of training trajectories. Actions are shared among queries with similar observed progress by tree leaves. Dynamic

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

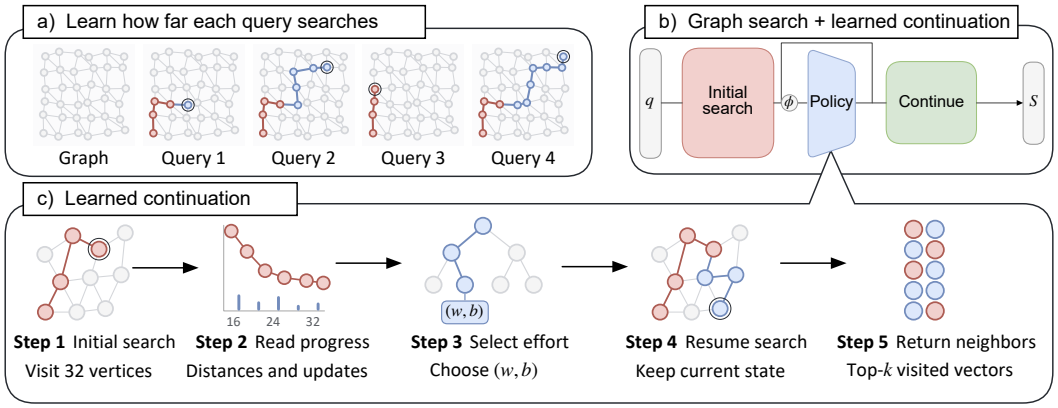


Figure 1: TraCANN selects frontier width w and total visit budget b from early progress, preserving search state. Coral/blue: initial/continuation visits; gray: unvisited vertices; rings: latest visit.

programming optimizes leaf actions simultaneously to minimize training cost subject to a constraint on the number of missing neighbors. We analyze the benefits of trading search effort and action granularity, and bound the cost of discrete leaf actions through a continuous relaxation. We reuse each training trajectory for multiple stopping budgets to obtain the measurements. The trained tree is used independently for each new query in its current search state.

We evaluate TraCANN on four publicly available ANN benchmarks against eleven other methods, including learned and distance-based stopping rules. At a 99% Recall@10 target, throughput is 1.60–2.72× that of the fastest measured baseline method on a single CPU core and 1.61–2.41× on eight CPU cores. Policies at this target complete 8.8–23.9% of queries in the first 32 visits. GloVe100 search lengths range from 24 to 2,456 visited vertices, showing that learned policies can generate both short and very long searches.

We make the following contributions: (i) a formulation of ANN search as a learned allocation of distance evaluations subject to an aggregate recall constraint, (ii) a trajectory-conditioned policy with reusable measurements and optimal action assignment for a given tree partition, (iii) an analysis of effort exchanges, allocation granularity, and cost sensitivity, and (iv) a CPU implementation of ANN search, evaluated against eleven ANN methods at three recall thresholds.

2 Related Work

Learned retrieval and adaptive computation. Retrieval serves two purposes, model inference and representation learning, in retrieval-augmented pre-training and ANN-based negative sampling (Guu et al., 2020; Xiong et al., 2021). Embedding adaptation and variable dimension representations improve the quality or cost of similarity search (Zeighami et al., 2025; Kusupati et al., 2022). The question we focus on is the computation cost in searching a fixed representation space. Other similar choices in learned computation include state skipping, recurrent refinement, and early exit which depend on the input (Campos et al., 2018; Dehghani et al., 2019; Bolukbasi et al., 2017), CALM which allocates language model depth given an output quality constraint (Schuster et al., 2022), and active fine tuning which spends more computation where it expects to improve (Hübotter et al., 2025). Self-RAG decides whether to retrieve or not in the context of generation (Asai et al., 2024). TraCANN makes an allocation decision within retrieval, using search progress and aggregate neighbor recovery to decide how many distance computations to do.

Graph traversal and learned indexes. Index construction decides which alternatives a search is allowed to explore. Indexes for HNSW, NSG, and Vamana are different types of proximity graphs (Malkov & Yashunin, 2020; Fu et al., 2019; Subramanya et al., 2019). Theory connects the geometry of graphs to search (Prokhorenkova & Shekhovtsov, 2020). CSPG reduces exploration via routing over graph partitions (Yang et al., 2024). Structured cells and posting lists offer alternative index organizations (Babenko & Lempitsky, 2012; Chen et al., 2021), including those for deep descriptors

and redundant assignments (Babenko & Lempitsky, 2016; Sun et al., 2023). Learned indexes adapt routing representations (Baranchuk et al., 2019), partitions of space (Dong et al., 2020), priority of clusters (Chiu et al., 2020), and balancing of buckets (Gupta et al., 2022). TraCANN keeps the graph fixed and learns how to continue a search in progress. Instead of learning how to construct a graph, decisions over which alternatives to retain and over how many vertices to visit are made while pursuing the same retrieval objective.

Efficient distance evaluation. Another strategy is to reduce the cost of evaluating each candidate score. Product quantization and optimized product quantization estimate distances using compact codes (Jégou et al., 2011; Ge et al., 2013), while FastScan and Quicker ADC speed up the distance lookups using SIMD (André et al., 2015; 2021). ScaNN tailors quantization for retrieval and LoRANN estimates candidate scores using reduced-rank regression (Guo et al., 2020; Jääsaari et al., 2024). RaBitQ uses randomized encoding and analyzes its error theoretically (Gao & Long, 2024; Gao et al., 2025). SVS and SymphonyQG add compressed scoring to graph search (Aguerrebere et al., 2023; Gou et al., 2025). FINGER, ADSampling, and probabilistic kernels reduce the cost of running an individual distance or angle comparison (Chen et al., 2023; Gao & Long, 2023; Lu et al., 2026). We use the graph and edge encodings from SymphonyQG in our implementation. The policy determines which searches get additional full-vector distance evaluations.

Adaptive search termination. Query-adaptive search termination is already addressed by AdaptNN, which predicts a stopping count based on initial search features and continues the same search traversal (Li et al., 2020). Tao predicts search termination based on static features (Yang et al., 2021). DARTH predicts recall during search on graphs (Chatzakis et al., 2025). For navigable graphs, Distance Adaptive Beam Search applies a distance-based search termination rule (Al-Jazzazi et al., 2025). TraCANN jointly fits continuation actions, using the costs and neighbor recovery measured on training queries. Its average recall constraint defines a shared budget for missing neighbors across query groups based on their features. Fitting allocates this budget via joint width and visit decisions using width-conditioned trajectories to measure available actions.

3 Method

3.1 Retrieval Objective and Search Actions

Let $X = \{x_1, \ldots, x_n\} \subset \mathbb{R}^d$ be the set of database representations and $\mathcal{Q}_{\text{tr}} = \{q_1, \ldots, q_m\}$ be the set of training queries. For squared Euclidean distance, let $N_k(q)$ be the set of k-nearest database identifiers and $S(q)$ be the set of returned identifiers. Training recall is

$$\text{Recall}(\mathcal{Q}_{\text{tr}}) = \frac{1}{mk} \sum_{i=1}^{m} |S(q_i) \cap N_k(q_i)|. \tag{1}$$

We learn a policy π that maps the progress of an ongoing graph search to an action $a = (w, b)$. Parameter w is the frontier capacity: the maximum number of candidates to retain. Parameter b is the visit budget, the maximum number of distinct vertices to visit (including initial traversal). To visit a vertex, one computes its full vector distance and updates the set of returned neighbors. We use quantized edge estimates to identify candidates for future visits.

Let q_i be the training query, $t_i(a)$ be the cost of action a, and

$$u_i(a) = k - |S(q_i, a) \cap N_k(q_i)| \tag{2}$$

be the number of missing neighbors. Given a training recall target ρ and search features $\phi(q_i)$, the objective of fitting is

$$\min_{\pi} \sum_{i=1}^{m} t_i\big(\pi(\phi(q_i))\big) \quad \text{subject to} \quad \sum_{i=1}^{m} u_i\big(\pi(\phi(q_i))\big) \le B, \qquad B = \lfloor mk(1-\rho) \rfloor. \tag{3}$$

Each recovered neighbor reduces the number of missing neighbors by one regardless of which query it was recovered from. The time required to recover that neighbor depends on the continuation. Suppose two queries exchange computation. Shortening the first search saves c but reduces the

number of neighbors recovered by r, while extending the second search spends c' but recovers r' neighbors. The total effect of the exchange is

$$\Delta\,\mathrm{Recall} = \frac{r' - r}{mk}, \qquad \Delta\mathrm{Cost} = c' - c. \tag{4}$$

Whenever $r' \geq r$ and $c' < c$, the exchange decreases training cost without decreasing training recall. A cost-minimizing allocation has no such feasible exchange. The comparisons are used to learn the policy. At inference, $\pi(\phi(q))$ independently chooses the continuation for each query. Appendix C.1 generalizes this argument to leaf assignment and recall slack.

3.2 Search Progress and Continuation

The frontier starts at the graph entry vertex with $w_0 = 64$ frontier capacity. A sorted frontier, a set of visited nodes, and a partial ranking of the best 16 visited nodes are maintained. If a node's identifier is seen multiple times, the frontier is advanced without further distance evaluation. After $h = 32$ different visits, we provide the policy with 20 features summarizing current distances, local graph geometry, and recent progress.

Distance ratios and concentration statistics characterize the neighborhood of the query. Changes at visits 16, 24, and 32 record recent progress. Each database vertex has a local scale

$$r(x) = \frac{1}{8} \sum_{y \in \mathcal{N}_8(x)} \|x - y\|_2^2, \tag{5}$$

where $\mathcal{N}_8(x)$ is its eight closest outgoing graph neighbors. Ratios of query distances to these scales characterize progress relative to local geometry. Appendix A defines the features and Appendix C.8 proves their scaling and distance-concentration interpretation.

A decision tree evaluates $\phi(q)$ once and returns (w, b). The frontier is resized to keep the current frontier elements, visited identifiers, and retrieved results. Traversal continues until the budget is reached or the frontier is empty. Queries that finish before visit 32 are returned directly. Results are updated before stopping to allow the last visit to improve retrieval. Algorithm 1 in Appendix B.4 describes the procedure.

3.3 Policy Learning from Search Trajectories

Fitting requires the costs and retrieval results of the available continuations on $\mathcal{Q}_{\mathrm{tr}}$. For each training query and candidate width, we continue the shared initial state until its frontier is exhausted, measuring costs and missing neighbors at checkpoints corresponding to various visit numbers. A single trajectory therefore produces measurements for many stopping budgets.

Lemma 1 (Trajectory reuse)**.** *Fix a query, a deterministic graph implementation, and a width after the shared prefix. For any budget $b \geq 32$, stopping returns the result list recorded after b distinct visits on the corresponding full trajectory. If the frontier is exhausted earlier, both procedures return the list at exhaustion.*

When the width is fixed, the visited sets are nested as the budget increases. Under a common distance ordering of reference and retrieved neighbors, let $\tau_{ij}(w)$ be the visit at which reference neighbor j is first reached, taking value ∞ if not reached. If action (w, b) performs $v_i(w, b)$ visits, then

$$u_i(w, b) = \sum_{j=1}^{k} \mathbf{1}\{\tau_{ij}(w) > v_i(w, b)\}. \tag{6}$$

Neighbor recovery occurs only at these discovery visits. Intervening visits consume computation before another neighbor is recovered. Appendix C.2 proves both these claims and computes savings from collecting all budgets along a single trajectory. We have 2,454 actions on SIFT1M, DEEP-1M and GIST1M and 3,706 on GloVe100 on our dense grid.

We group training queries by observed progress in tree splits. Each split minimizes the sum of its children's best losses using $t_i(a) + \eta u_i(a)/k$. We create depth-12 partitions over a coarse action grid

and then refine their leaves up to two levels with the dense grid. Details of the penalty sweep, split search, and measured costs are in Appendix B.

Let I be a subset of the query set and $\mathcal{A}$ be a common candidate set. Define

$$H_\lambda(I) = \min_{a \in \mathcal{A}} \sum_{i \in I} \big(t_i(a) + \lambda u_i(a)\big), \qquad \lambda = \eta/k. \tag{7}$$

A split $I = I^- \dot\cup I^+$ has gain

$$G_\lambda(I^-, I^+) = H_\lambda(I) - H_\lambda(I^-) - H_\lambda(I^+) \geq 0. \tag{8}$$

This gain is strictly positive if and only if the children have no common minimizing action. A split is thus useful (i.e. improves the partition) if and only if its two children prefer different continuations at the same recovery price. This statement is proved in Appendix C.3, along with a discussion of accumulated split gains and their relation to action-sharing costs.

3.4 Action Allocation under a Recall Constraint

The cost and misses of each action for training queries I_ℓ to leaf ℓ are aggregated as

$$C_\ell(a) = \sum_{i \in I_\ell} t_i(a), \qquad M_\ell(a) = \sum_{i \in I_\ell} u_i(a). \tag{9}$$

Training queries that complete before policy evaluation have fixed cost C_0 and misses M_0. The minimum possible training cost for partition $\mathcal{P} = \{I_1, \ldots, I_L\}$ is

$$V_\mathcal{P}(B) = C_0 + \min_{a_\ell \in \mathcal{A}_\ell \,,\, \sum_\ell M_\ell(a_\ell) \leq B - M_0} \sum_{\ell=1}^{L} C_\ell(a_\ell). \tag{10}$$

We keep only the lowest cost for each attainable miss count, and discard actions with higher cost and no fewer misses. The penalized losses used to build the tree do not suffice to solve this discrete assignment problem. Consider a leaf with miss-cost pairs $(0, 10)$, $(1, 6)$, and $(2, 0)$. Under a budget of at most one miss, $(1, 6)$ is the cheapest feasible action, but it never minimizes $C + \lambda M$ for any nonnegative λ. Dynamic programming over integer miss counts keeps choices like this. Let $F_\ell(s)$ be the minimum measured cost of assigning the first ℓ leaves with exactly s misses. Starting with $F_0(0) = 0$ and $F_0(s) = \infty$ for $s > 0$, compute

$$F_\ell(s) = \min_{a \in \mathcal{A}_\ell \,,\, M_\ell(a) \leq s} \big\{ F_{\ell-1}\big(s - M_\ell(a)\big) + C_\ell(a) \big\}. \tag{11}$$

Backtracking from the minimum over $0 \leq s \leq B - M_0$ gives the assigned actions. With at most A retained actions per leaf, the recurrence takes $O(L(B+1)A)$ time, $O(B+1)$ working storage for costs, and $O(L(B+1))$ storage for backpointers.

Proposition 1 (Optimal action assignment)**.** *For a fixed partition, finite action sets, nonnegative measured costs, and integer miss counts, Equation (11) gives a minimum-cost feasible assignment whenever one exists. Refining the partition cannot increase the minimum, when the per-query costs, miss counts, available actions, and miss budget do not change.*

To make the refinement comparison explicit, fix a common action set and the cost and miss arrays. Write $\mathcal{P}_1$ for a single leaf and $\mathcal{P}_{\text{ind}}$ for one leaf per active query. Every refinement $\mathcal{P}'$ of $\mathcal{P}$ satisfies

$$V_{\mathcal{P}_{\text{ind}}}(B) \leq V_{\mathcal{P}'}(B) \leq V_\mathcal{P}(B) \leq V_{\mathcal{P}_1}(B). \tag{12}$$

Giving each child its parent's action preserves the total cost and recall, while allowing different child actions increases the set of possible allocations. Appendix C.4 proves both results.

3.5 Neighbor Recovery Costs and Allocation Granularity

The dual formulation is obtained by assigning a price $\lambda \geq 0$ to each missing neighbor:

$$D_\mathcal{P}(\lambda) = C_0 + \lambda(M_0 - B) + \sum_{\ell=1}^{L} \min_{a \in \mathcal{A}_\ell} \{C_\ell(a) + \lambda M_\ell(a)\} \leq V_\mathcal{P}(B). \tag{13}$$

When there are two leaf actions, retrieval of r additional neighbors at extra cost c improves the priced objective if and only if $c < \lambda r$, and this provides a common scale to compare the gains from retrieval across queries. The tree construction penalty is $\lambda = \eta/k$.

Proposition 2 (Allocation granularity)**.** *Let Equation (10) be feasible and suppose that every $\mathcal{A}_\ell$ is finite. Let $\Delta_\ell = \max_a C_\ell(a) - \min_a C_\ell(a)$. Then*

$$0 \le V_{\mathcal{P}}(B) - \max_{\lambda \ge 0} D_{\mathcal{P}}(\lambda) \le \max_{\ell} \Delta_\ell. \tag{14}$$

We show that the optimal value of a linear program with convex combinations of actions is equal to the dual bound. An optimal extreme point mixes actions in at most one leaf. Rounding this leaf to its lower-miss action does not hurt recall, and increases cost by at most the cost span of this leaf. If all query action cost spans are at most H, then the gap per query is at most $H \max_\ell |I_\ell|/m$. See Appendix C.5 for the derivation of this result and the corresponding assignment-specific bound. We study the stability of the algorithm to perturbations of the costs in Appendix C.7.

The same relaxation gives the cost of changing the recall target. Let $W_{\mathcal{P}}(B) = \max_{\lambda \ge 0} D_{\mathcal{P}}(\lambda)$ and let λ_j be the optimal price at a feasible budget B_j. If $B_1 < B_2$, then

$$\lambda_2(B_2 - B_1) \le W_{\mathcal{P}}(B_1) - W_{\mathcal{P}}(B_2) \le \lambda_1(B_2 - B_1). \tag{15}$$

The price of additional recovery is nondecreasing as the budget for misses tightens. In the fractional allocation, each additional unit of recovery is given to the cheapest improvement on the action frontiers. This gives an explicit relation between the recall target and the decision to continue individual searches. The frontier construction and bounds for integer assignments are in Appendix C.6.

3.6 CPU Implementation

TraCANN employs the graph builder, edge codes, and SIMD scorer from the RaBitQ-Library 0.3.6 (Gou et al., 2025; Gao et al., 2025). It incurs $\Theta(dv_i)$ distance work for v_i visits (one full-vector distance evaluation per unique visited vertex), plus at most 14 comparisons from a single tree evaluation. SIFT descriptors employ lossless byte storage and AVX-512 VNNI dot products, whereas the other datasets use FP32. OpenMP dynamically schedules queries with private search state. Appendix C.9 contains details of traversal and storage bounds and Appendix C.10 contains details of byte distance calculation.

4 Experiments

4.1 Datasets and Evaluation Setup

We evaluate on four public ANN benchmarks that were used in previous retrieval studies (Aumüller et al., 2020; Gao & Long, 2024; Gou et al., 2025; Jääsaari et al., 2024). The sizes and representations of the datasets are shown in Table 1. SIFT1M and GIST1M contain image descriptors (Lowe, 2004; Oliva & Torralba, 2001), DEEP-1M contains 256-dimensional image embeddings, and GloVe100 contains learned word embeddings (Pennington et al., 2014). 4,400 test queries are evaluated against the entire database.

Table 1: ANN benchmarks. GloVe100 is unit-normalized (squared Euclidean distance); other datasets retain their original FP32 vectors.

Dataset	Database size	Dimension	Test queries	Vector type
SIFT1M	1,000,000	128	1,200	Image descriptors
DEEP-1M	1,000,000	256	1,000	Image embeddings
GIST1M	1,000,000	960	1,000	Image descriptors
GloVe100	1,183,514	100	1,200	Word embeddings

Queries for training are disjoint from the test sets in Table 1. The trajectories and neighbor labels determine the tree partitions and continuation actions. For each dataset and recall target, the policy learned is kept fixed for the entirety of testing. Each test query chooses its action based on its traversal features. Recall@10 measures recovery of reference identifiers. For each threshold in $\{95\%, 99\%, 99.5\%\}$, we compare the fastest measured configuration that attains it. Test recall is reported in Appendix E along with throughput.

All methods run on one or eight pinned physical cores within one NUMA node of an Intel Xeon Platinum 8352V, in batches of 128 queries. QPS is computed from median query duration after warmup, including native search, binding calls, and result production but excluding index loading, policy fitting, and recall computation. Appendix D specifies repetitions and CPU placement.

External methods. The eleven comparison methods are HNSW (Malkov & Yashunin, 2020), SymphonyQG (Gou et al., 2025), SVS/LVQ (Aguerrebere et al., 2023), LoRANN (Jääsaari et al., 2024), RaBitQ-IVF (Gao et al., 2025), ScaNN (Guo et al., 2020), OPQ+FastScan (André et al., 2015; Douze et al., 2024), HNSW+KS2 (Lu et al., 2026), AdaptNN (Li et al., 2020), DARTH (Chatzakis et al., 2025), and Distance Adaptive Beam Search (DABS) (Al-Jazzazi et al., 2025). We use the publicly available native implementation of each backend. The parameter search covers 80 construction configurations over 44 method-dataset combinations, followed by up to two local refinements near each recall target. KS2 is considered in the single-core comparisons.

RaBitQ-IVF includes the maintained library's quantized and 32-bit raw-vector modes. SymphonyQG uses RaBitQ-Library with PiPNN initialization; see Appendix D for versions and build options. Our source graph uses degree 32 and construction effort 400 on SIFT1M, and degree 64 and construction effort 800 elsewhere. Both graph configurations are among the comparison candidates for SymphonyQG.

4.2 Throughput across Recall Targets

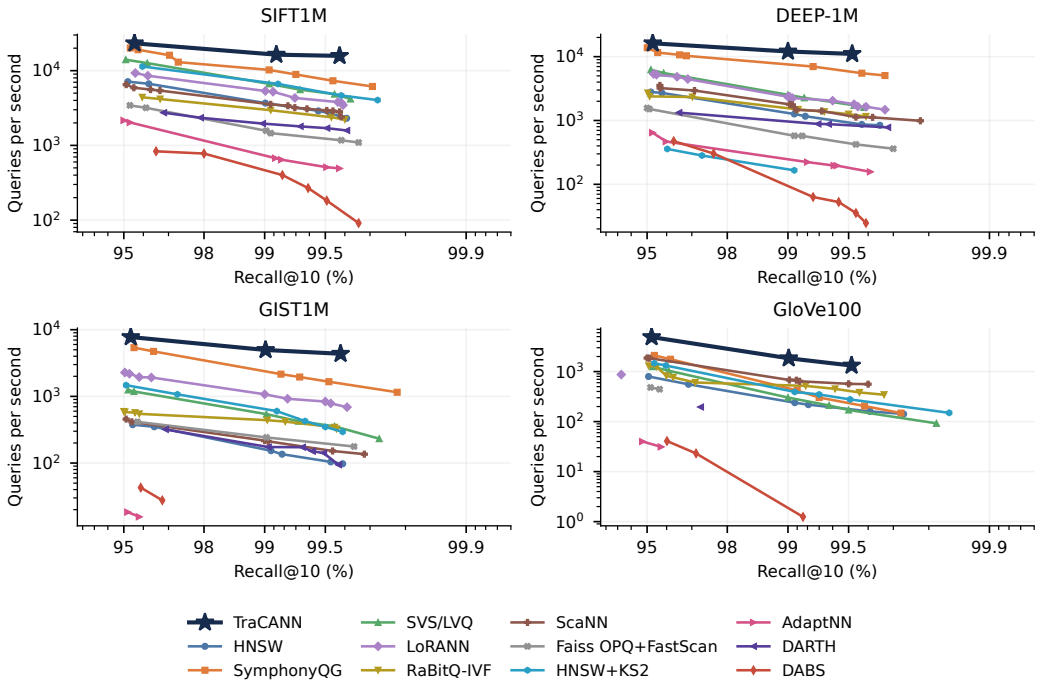


Figure 2: Single-core QPS versus achieved Recall@10, on a log miss-rate axis. Stars: TraCANN policies; curves: confirmed, non-dominated baseline settings.

TraCANN has the highest throughput among the eleven external methods at each of the three recall thresholds on all four datasets (Figure 2). The best baseline varies by dataset and recall threshold. SymphonyQG performs best on SIFT1M, DEEP-1M and GIST1M, while on GloVe100, it performs best at 95% recall and ScaNN at 99% and 99.5%.

At 99% recall, TraCANN is 1.60, 1.72, 2.30, and 2.72× as fast as the strongest baseline on SIFT1M, DEEP-1M, GIST1M, and GloVe100, respectively. The results for every backend are shown in Table 2. Among adaptive termination methods, DARTH is the fastest on the first three datasets. DABS reaches 99.158% recall at 1.2 QPS on GloVe100, while AdaptNN and DARTH do not reach the threshold.

Table 2: Single-core QPS at Recall@10 $\geq 99\%$ (fastest qualifying settings). Ratios use the strongest baseline; N/A: no qualifying setting.

Method	SIFT1M	DEEP-1M	GIST1M	GloVe100
HNSW	3,685	1,263	153	239
SymphonyQG	10,236	6,997	2,155	419
SVS/LVQ	6,661	2,528	542	303
LoRANN	5,345	2,356	1,075	N/A
RaBitQ-IVF	2,958	1,490	439	518
ScaNN	3,542	1,789	217	679
OPQ+FastScan	1,572	576	242	N/A
HNSW+KS2	6,647	166	601	392
AdaptNN	674	223	N/A	N/A
DARTH	1,796	883	174	N/A
DABS	401	63.2	N/A	1.2
TraCANN	**16,369**	**12,056**	**4,966**	**1,846**
Throughput ratio	1.60×	1.72×	2.30×	2.72×

The single-core benefit continues at the other thresholds: throughput is 2.00–2.62× that of the fastest baseline at 99.5% recall and 1.15–2.32× at 95%. At 99%, GIST1M and GloVe100 see the most benefit, even though they differ in both dimensionality and representation. Section 4.4 investigates how our policies divide the computation over their queries.

4.3 Parallel Batch Search

Table 3: Eight-core QPS at Recall@10 $\geq 99\%$. Baselines: SymphonyQG except ScaNN on GloVe100. Scaling: eight-core/single-core QPS.

Dataset	Ours	Best baseline	Ratio	Core scaling
SIFT1M	123,036	76,593	1.61×	7.52×
DEEP-1M	89,336	54,194	1.65×	7.41×
GIST1M	33,452	14,880	2.25×	6.74×
GloVe100	12,052	5,004	2.41×	6.53×

At eight cores, the same policies and batch size lead to throughput ratios 1.61, 1.65, 2.25, and 2.41 against the strongest baseline (Table 3). TraCANN scales 6.53–7.52× from 1 to 8 cores. DARTH and DABS use eight serial workers in a process pool, including dispatch and result collection in the measured time; the other parallel methods use native threads.

Dynamic scheduling assigns long and short traversals to workers. Policy evaluation is read-only, with no synchronization across queries. Figure 5 and Table 8 show eight-core comparisons. TraCANN achieves 1.15–4.75× the throughput of the strongest baseline across all 24 dataset, recall target, and core count combinations.

4.4 Search Effort and Policy Size

Figure 3 shows full-vector distance evaluations per test query. At 99% recall, median visits are 76, 56, 92, and 176 on SIFT1M, DEEP-1M, GIST1M, and GloVe100, respectively; corresponding 95th percentiles are 152, 128, 256.2, and 1,488.4. The 24–2,456 visits on GloVe100 illustrate the range of search lengths selected from early traversal features.

Between 8.8% and 23.9% of queries finish within the first 32 visits; the rest follow tree-selected continuation actions. Mean visit counts are 81.2, 69.2, 104.6, and 421.6. Each visit computes one full-vector distance, with quantized edge comparisons guiding navigation between visits.

The fitted trees have 291–410 leaves and require at most 14 threshold comparisons. Tree-node payloads occupy 11.3–16.0 KiB; local scales use 3.8–4.5 MiB. Table 12 reports storage and process memory.

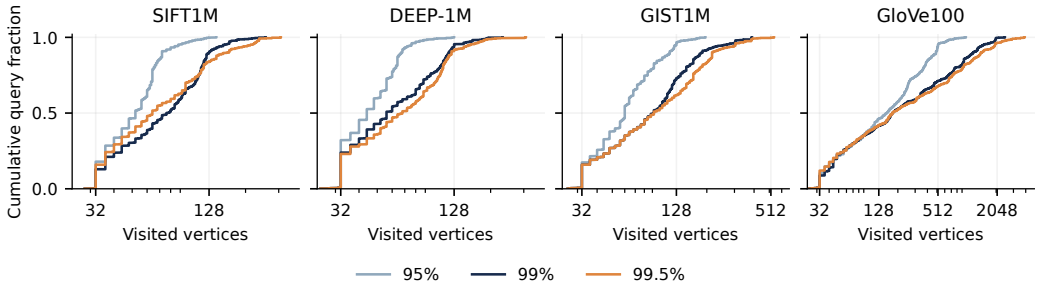


Figure 3: Test-query visits by target recall; one full-vector distance evaluation per distinct visit.

Table 4: Search effort and tree size at 99% recall. Stop-by-32 includes frontier exhaustion; percentiles use linear interpolation.

Dataset	Mean visits	Median	95th percentile	Stop by 32	Leaves
SIFT1M	81.2	76	152.0	12.9%	309
DEEP-1M	69.2	56	128.0	23.9%	291
GIST1M	104.6	92	256.2	15.9%	343
GloVe100	421.6	176	1488.4	8.8%	410

4.5 Joint Width and Visit Allocation

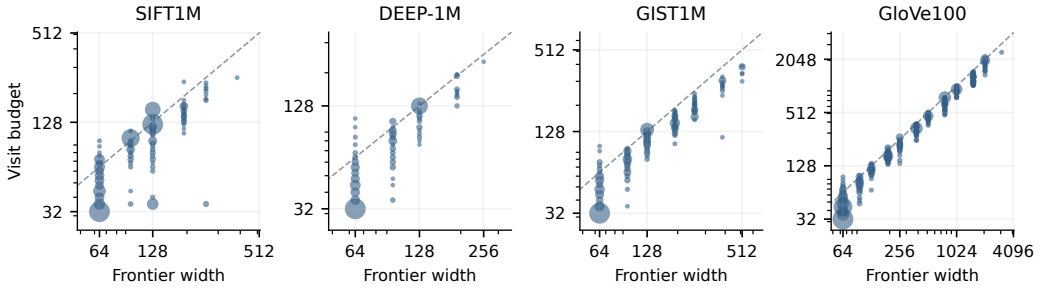


Figure 4: Width–budget assignments on active training queries for the 99% policies. Marker area indicates the number of training queries assigned each action; dashed lines indicate equal width and visit budget. Queries ending before the prefix are omitted.

The learned policies use 73, 51, 90, and 208 distinct actions on SIFT1M, DEEP-1M, GIST1M, and GloVe100, respectively (Figure 4). Median widths among active training queries are 96, 64, 128, and 192. Across datasets, 28.3%–53.0% of these queries retain width 64; the others receive wider frontiers, reaching 3072 on GloVe100.

Queries assigned the same width can receive different visit budgets. On GloVe100, 92.7% of active training queries receive budgets below frontier capacity. Width limits the number of alternatives retained, while the visit budget limits how far search explores them. Joint fitting thus retains alternative routes without requiring an equally long sequence of full-vector distance evaluations.

5 Conclusion

TraCANN learns how far to continue retrieval from training trajectories under an aggregate recall constraint. It learns a policy to independently determine how to continue retrieval based on each query's early search progress. We analyze the relationships between partition learning, marginal recovery costs, and discrete action allocation. Our policies reach the highest throughput across the evaluated recall targets on four public ANN datasets compared with eleven ANN methods.

REFERENCES

Cecilia Aguerrebere, Ishwar Singh Bhati, Mark Hildebrand, Mariano Tepper, and Theodore Willke. Similarity search in the blink of an eye with compressed indices. *Proceedings of the VLDB Endowment*, 16(11):3433–3446, 2023. doi: 10.14778/3611479.3611537.

Yousef Al-Jazzazi, Haya Diwan, Jinrui Gou, Cameron Musco, Christopher Musco, and Torsten Suel. Distance adaptive beam search for provably accurate graph-based nearest neighbor search. In *Advances in Neural Information Processing Systems*, volume 38, 2025.

Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. Cache locality is not enough: High-performance nearest neighbor search with product quantization fast scan. *Proceedings of the VLDB Endowment*, 9(4):288–299, 2015. doi: 10.14778/2856318.2856324.

Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. Quicker ADC: Unlocking the hidden potential of product quantization with SIMD. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(5):1666–1677, 2021. doi: 10.1109/TPAMI.2019.2952606.

Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. Self-RAG: Learning to retrieve, generate, and critique through self-reflection. In *International Conference on Learning Representations*, 2024.

Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, 87:101374, 2020. doi: 10.1016/j.is.2019.02.006.

Artem Babenko and Victor Lempitsky. The inverted multi-index. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3069–3076, 2012. doi: 10.1109/CVPR.2012.6248038.

Artem Babenko and Victor Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2055–2063, 2016.

Dmitry Baranchuk, Dmitry Persiyanov, Anton Sinitsin, and Artem Babenko. Learning to route in similarity graphs. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 475–484, 2019.

Tolga Bolukbasi, Joseph Wang, Ofer Dekel, and Venkatesh Saligrama. Adaptive neural networks for efficient inference. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 527–536, 2017.

Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. Improving language models by retrieving from trillions of tokens. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 2206–2240, 2022.

Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

Víctor Campos, Brendan Jou, Xavier Giró-i Nieto, Jordi Torres, and Shih-Fu Chang. Skip RNN: Learning to skip state updates in recurrent neural networks. In *International Conference on Learning Representations*, 2018.

Manos Chatzakis, Yannis Papakonstantinou, and Themis Palpanas. DARTH: Declarative recall through early termination for approximate nearest neighbor search. *Proceedings of the ACM on Management of Data*, 3(4):242, 2025. doi: 10.1145/3749160. Presented at SIGMOD 2026.

Patrick H. Chen, Wei-Cheng Chang, Jyun-Yu Jiang, Hsiang-Fu Yu, Inderjit S. Dhillon, and Cho-Jui Hsieh. FINGER: Fast inference for graph-based approximate nearest neighbor search. In *Proceedings of the ACM Web Conference*, pp. 3225–3235, 2023. doi: 10.1145/3543507.3583318.

Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. SPANN: Highly-efficient billion-scale approximate nearest neighborhood search. In *Advances in Neural Information Processing Systems*, volume 34, pp. 5199–5212, 2021.

Chih-Yi Chiu, Amorntip Prayoonwong, and Yin-Chih Liao. Learning to index for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(8):1942–1956, 2020. doi: 10.1109/TPAMI.2019.2907086.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. Universal transformers. In *International Conference on Learning Representations*, 2019.

Yihe Dong, Piotr Indyk, Ilya Razenshteyn, and Tal Wagner. Learning space partitions for nearest neighbor search. In *International Conference on Learning Representations*, 2020.

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

Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proceedings of the VLDB Endowment*, 12(5):461–474, 2019. doi: 10.14778/3303753.3303754.

Jianyang Gao and Cheng Long. High-dimensional approximate nearest neighbor search: with reliable and efficient distance comparison operations. *Proceedings of the ACM on Management of Data*, 1 (2):137:1–137:27, 2023. doi: 10.1145/3589282.

Jianyang Gao and Cheng Long. RaBitQ: Quantizing high-dimensional vectors with a theoretical error bound for approximate nearest neighbor search. *Proceedings of the ACM on Management of Data*, 2(3):167:1–167:27, 2024. doi: 10.1145/3654970.

Jianyang Gao, Yutong Gou, Yuexuan Xu, Yongyi Yang, Cheng Long, and Raymond Chi-Wing Wong. Practical and asymptotically optimal quantization of high-dimensional vectors in euclidean space for approximate nearest neighbor search. *Proceedings of the ACM on Management of Data*, 3(3): 202:1–202:26, 2025. doi: 10.1145/3725413.

Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2946–2953, 2013.

Yutong Gou, Jianyang Gao, Yuexuan Xu, and Cheng Long. SymphonyQG: Towards symphonious integration of quantization and graph for approximate nearest neighbor search. *Proceedings of the ACM on Management of Data*, 3(1):80:1–80:26, 2025. doi: 10.1145/3709730.

R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9): 1563–1581, 1966. doi: 10.1002/j.1538-7305.1966.tb01709.x.

Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 3887–3896, 2020.

Gaurav Gupta, Tharun Medini, Anshumali Shrivastava, and Alexander J. Smola. BLISS: A billion scale index using iterative re-partitioning. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 486–495, 2022. doi: 10.1145/3534678.3539414.

Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. Retrieval augmented language model pre-training. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 3929–3938, 2020.

Jonas Hübotter, Sascha Bongni, Ido Hakimi, and Andreas Krause. Efficiently learning at test-time: Active fine-tuning of LLMs. In *International Conference on Learning Representations*, 2025.

Elias Jääsaari, Ville Hyvönen, and Teemu Roos. LoRANN: Low-rank matrix factorization for approximate nearest neighbor search. In *Advances in Neural Information Processing Systems*, volume 37, pp. 102121–102153, 2024.

Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011. doi: 10.1109/TPAMI.2010.57.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, pp. 6769–6781, 2020. doi: 10.18653/v1/2020.emnlp-main.550.

Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Generalization through memorization: Nearest neighbor language models. In *International Conference on Learning Representations*, 2020.

Aditya Kusupati, Gantavya Bhatt, Aniket Rege, Matthew Wallingford, Aditya Sinha, Vivek Ramanujan, William Howard-Snyder, Kaifeng Chen, Sham Kakade, Prateek Jain, and Ali Farhadi. Matryoshka representation learning. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

Elizaveta Levina and Peter J. Bickel. Maximum likelihood estimation of intrinsic dimension. In *Advances in Neural Information Processing Systems*, volume 17, 2004.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, 2020.

Conglong Li, Minjia Zhang, David G. Andersen, and Yuxiong He. Improving approximate nearest neighbor search through learned adaptive early termination. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 2539–2554, 2020. doi: 10.1145/3318464.3380600.

David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004. doi: 10.1023/B:VISI.0000029664.99615.94.

Kejing Lu, Chuan Xiao, and Yoshiharu Ishikawa. Probabilistic kernel function for fast angle testing. In *International Conference on Learning Representations*, 2026.

Yury A. Malkov and Dmitry A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, 2020. doi: 10.1109/TPAMI.2018.2889473.

Aude Oliva and Antonio Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International Journal of Computer Vision*, 42(3):145–175, 2001.

Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pp. 1532–1543, 2014. doi: 10.3115/v1/D14-1162.

Liudmila Prokhorenkova and Aleksandr Shekhovtsov. Graph-based nearest neighbor search: From practice to theory. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 7803–7813, 2020.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 8748–8763, 2021.

Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, pp. 3982–3992, 2019. doi: 10.18653/v1/D19-1410.

Tal Schuster, Adam Fisch, Jai Gupta, Mostafa Dehghani, Dara Bahri, Vinh Q. Tran, Yi Tay, and Donald Metzler. Confident adaptive language modeling. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. DiskANN: Fast accurate billion-point nearest neighbor search on a single node. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

Philip Sun, David Simcha, Dave Dopson, Ruiqi Guo, and Sanjiv Kumar. SOAR: Improved indexing for approximate nearest neighbor search. In *Advances in Neural Information Processing Systems*, volume 36, pp. 3189–3204, 2023. doi: 10.52202/075280-0141.

Lee Xiong, Chenyan Xiong, Ye Li, Kwok-Fung Tang, Jialin Liu, Paul Bennett, Junaid Ahmed, and Arnold Overwijk. Approximate nearest neighbor negative contrastive learning for dense text retrieval. In *International Conference on Learning Representations*, 2021.

Kaixiang Yang, Hongya Wang, Bo Xu, Wei Wang, Yingyuan Xiao, Ming Du, and Junfeng Zhou. Tao: A learning framework for adaptive nearest neighbor search using static features only. *arXiv preprint arXiv:2110.00696*, 2021.

Ming Yang, Yuzheng Cai, and Weiguo Zheng. CSPG: Crossing sparse proximity graphs for approximate nearest neighbor search. In *Advances in Neural Information Processing Systems*, volume 37, pp. 103076–103100, 2024. doi: 10.52202/079017-3275.

Sepanta Zeighami, Zac Wellmer, and Aditya Parameswaran. NUDGE: Lightweight non-parametric fine-tuning of embeddings for retrieval. In *International Conference on Learning Representations*, 2025.

A Search Features

Let $d_j^{(h)}$ be the jth smallest squared distance among the vertices visited by step h, with identifier $x_j^{(h)}$. The features are evaluated at step $h = 32$. Let $D_j = \max\{d_j^{(32)}, 10^{-12}\}$ and $R_j = \max\{r(x_j^{(32)}), 10^{-12}\}$. We define

$$\bar{D} = \frac{1}{16}\sum_{j=1}^{16} D_j, \qquad \sigma_D^2 = \frac{1}{16}\sum_{j=1}^{16}(D_j - \bar{D})^2, \tag{16}$$

$$\bar{R} = \frac{1}{8}\sum_{j=1}^{8} R_j, \qquad \sigma_R^2 = \frac{1}{8}\sum_{j=1}^{8}(R_j - \bar{R})^2. \tag{17}$$

The first thirteen features describe distances and local scales, while the last seven describe search progress and frontier activity. Their definitions in the order implemented are given in Table 5 using one-based indices.

Table 5: Features measured after 32 distinct vertex visits. All distances are squared Euclidean distances.

Index	Definition	Measurement
1	$\log D_1$	Nearest visited distance
2	$\log D_{10}$	Tenth visited distance
3	D_1/D_{10}	Relative nearest distance
4	D_2/D_{10}	Relative second distance
5	D_5/D_{10}	Relative fifth distance
6	D_{16}/D_{10}	Spread beyond the result list
7	$-1/\min\{\frac{1}{15}\sum_{j=1}^{15}\log(D_j/D_{16}), -10^{-5}\}$	Concentration among 16 vertices
8	$-1/\min\{\frac{1}{9}\sum_{j=1}^{9}\log(D_j/D_{10}), -10^{-5}\}$	Concentration among 10 vertices
9	$\|q\|_2^2$	Query magnitude
10	D_{10}/R_1	Distance relative to nearest local scale
11	$D_{10}/\bar{R}$	Distance relative to mean local scale
12	$\sigma_R/\bar{R}$	Variation in local scale
13	$\sigma_D/\bar{D}$	Variation in visited distances
14	$d_{10}^{(16)}/D_{10}$	Improvement since visit 16
15	$d_{10}^{(24)}/D_{10}$	Improvement since visit 24
16	$d_1^{(16)}/D_1$	Nearest-distance improvement
17	$1 - \|T^{(24)} \cap T^{(32)}\|/10$	Turnover in the current top ten
18	$\widehat{d}_{\text{next}}/D_{10}$	Next frontier score
19	$(P - 32)/P$	Fraction of repeated pops
20	$(J^{(32)} - J^{(24)})/8$	Recent insertion rate

$T^{(h)}$ is the set of 10 closest identifiers after h visits. P is the number of frontier pops up to visit 32. $J^{(h)}$ is the number of insertions up to visit h. Feature 18 is the estimated score of the next unprocessed frontier element. This feature is set to 2 if there is no unprocessed frontier element. Features 7 and 8 are summaries of the concentration of the squared distances. All features are computed using the query, graph, and the current search state.

B Policy Construction

B.1 Action Sets and Cost Measurements

The coarse partition construction uses

$$\mathcal{W}_{\text{c}} = \{64, 128, 256, 512, 1024, 2048\}, \tag{18}$$

$$\mathcal{B}_{\text{c}} = \{32, 48, 64, 96, 128, 192, 256, 384, 512, 768, 1024, 1536, 2048, 3072, 4096, \infty\}. \tag{19}$$

GloVe100 also has width 4096. We keep $(64, 32)$ and actions with $b > 32$ where either $b = \infty$ or $w/4 \le b \le 1.5w$. We have 39 actions for the first three datasets and 45 for GloVe100.

Dense refinement uses

$$\mathcal{W}_{\mathrm{d}} = \{64, 96, 128, 192, 256, 384, 512, 768, 1024, 1536, 2048\}, \tag{20}$$

augmented with widths 3072 and 4096 for GloVe100. Its 762 stopping checkpoints are

$$\mathcal{B}_{\mathrm{d}} = \{32, 36, \ldots, 1024\} \cup \{1032, 1040, \ldots, 4096\} \cup \{4128, 4160, \ldots, 8192\} \cup \{\infty\}. \tag{21}$$

This action set preserves $(64, 32)$, plus actions with $b > 32$ satisfying either $b = \infty$ or $w/8 \le b \le 2w$, resulting in 2,454 actions for SIFT1M, DEEP-1M, and GIST1M, and 3,706 for GloVe100. Using budget $b = \infty$ allows search until frontier exhaustion, and this is signified by checkpoint value 65535.

For each width, two passes are made through the training query set $\mathcal{Q}_{\mathrm{tr}}$. Action measurements are taken on the second pass, taking all stopping checkpoints from one trajectory per training query. Timing begins before query rotation and frontier initialization. At each checkpoint, the ten returned identifiers are compared to the reference neighbors of the training query, and the time taken is subtracted from subsequent elapsed times. Checkpoint timing includes processing of outgoing edges at the corresponding visit. Online search does not process outgoing edges at its final visit when $b > 32$. These trajectory costs are the fitting objective, with the costs of each action capped at their 99th percentile across training queries. Reported QPS is based on complete online searches on test queries with unclipped timing measurements.

B.2 Partition Search

The coarse tree is built by evaluating 80 logarithmically spaced penalty values between 10 and 10^7. For each value, the loss of action a on query i is $t_i(a) + \eta u_i(a)/10$, where t_i is in microseconds. At each node, each feature provides 22 thresholds at equally spaced quantiles between 0.08 and 0.92. A split is eligible if at least 4 queries end up in each child. We select the split that minimizes the sum of the best action losses of the children, and only split if the loss decreases in comparison to the unsplit node. The maximum depth of coarse trees is 12.

For a requested training recall, we find the 14 coarse trees with action assignments closest to the total training recall. Integer-budget allocation chooses actions for each tree, and the lowest-cost feasible partition proceeds to dense refinement. Refinement performs up to two levels below each leaf and requires at least two training queries per child. Candidate splits are at boundaries between different observed feature values, with the lower value used as the threshold. Split evaluation uses the union of the 32 lowest-loss actions for the parent and the eight lowest-loss actions for each query inside it. Final allocation is over the full set of dense actions minus dominated ones.

B.3 Integer Allocation

Within each leaf ℓ, actions with the same miss count are reduced to their lowest-cost representative. The remaining actions can be sorted by increasing miss count, which allows for a linear pass to discard any action whose cost is at least as large as the best cost seen so far. The remaining actions are then used in Equation (11). The costs are in floating-point microseconds, the miss counts and budget indices are integers. The integer budget must be calculated by adding 10^{-7} before the floor operation to ensure the correct integer boundaries.

The recurrence stores the minimum cost and a backpointer to the action selected for each achievable miss count. Backtracking starts from the lowest cost terminal state satisfying the budget. Queries with empty frontier before visit 32 have fixed costs and misses outside the leaf assignment.

B.4 Online Search

Algorithm 1 retains the frontier, visited identifiers, and result list throughout the single policy decision. Candidates discarded in the prefix are discarded throughout the rest of the algorithm. The visit budget also includes the first 32 visits. If $b > 32$, the last visit updates the result list without needing to process outgoing edges. The choice of $b = 32$ occurs after the prefix has produced the policy features.

Algorithm 1 TraCANN query search

Require: Query q, graph G, local scales r, policy π
Initialize frontier F with the entry vertex and capacity 64
Initialize visited set V and best-16 result list R
$e \leftarrow 0$, $b \leftarrow \infty$
while F contains an unprocessed element **do**
    Remove the next identifier x from F
    if $x \notin V$ **then**
        Add x to V, increment e, evaluate $\|q - x\|_2^2$, update R
        if $e = b$ **then**
            break
        end if
        Score outgoing neighbors in groups of 32 and update F
        Retain progress statistics at visits 16 and 24
        if $e = 32$ **then**
            $(w, b) \leftarrow \pi(\phi(q, F, R, V, r))$
            Resize F to capacity w, preserving its contents
            if $b = 32$ **then**
                break
            end if
        end if
    end if
end while
return The ten closest identifiers in R

C Analysis and Proofs

In the allocation analysis, we fix the training set $\mathcal{Q}_{\mathrm{tr}}$, feature partition, candidate actions, cost and miss arrays. The costs are the training action measurements in (3). Training queries that are completed before policy evaluation contribute C_0 and M_0. For an assignment $\boldsymbol{a} = (a_1, \ldots, a_L)$, define

$$C(\boldsymbol{a}) = C_0 + \sum_{\ell=1}^{L} C_\ell(a_\ell), \qquad M(\boldsymbol{a}) = M_0 + \sum_{\ell=1}^{L} M_\ell(a_\ell). \tag{22}$$

Feasibility is defined by $M(\boldsymbol{a}) \le B$. The feature partition has $L \ge 1$ nonempty leaves, with finite nonempty action sets.

C.1 Recall Accounting and Effort Exchanges

Each query needs k neighbors, so Eq. (1) yields

$$\mathrm{Recall}(\mathcal{Q}_{\mathrm{tr}}) = 1 - \frac{M(\boldsymbol{a})}{mk}, \qquad \mathrm{Recall}(\mathcal{Q}_{\mathrm{tr}}) \ge \rho \iff M(\boldsymbol{a}) \le \lfloor mk(1-\rho) \rfloor. \tag{23}$$

The floor is due to the integrality of missing neighbor counts. It states that the recall requirement is a shared resource constraint with one unit per missed neighbor.

Lemma 2 (Feasible exchanges)**.** *Let $\boldsymbol{a}$ be feasible with slack $s = B - M(\boldsymbol{a})$. Replace its actions on a set of leaves J by $\widetilde{a}_\ell$, and write*

$$\Delta C_J = \sum_{\ell \in J} \big(C_\ell(\widetilde{a}_\ell) - C_\ell(a_\ell)\big), \qquad \Delta M_J = \sum_{\ell \in J} \big(M_\ell(\widetilde{a}_\ell) - M_\ell(a_\ell)\big). \tag{24}$$

The replacement is feasible if and only if $\Delta M_J \le s$. It improves cost if and only if $\Delta C_J < 0$.

Proof. The new objective and constraint values are $C(\boldsymbol{a}) + \Delta C_J$ and $M(\boldsymbol{a}) + \Delta M_J$, respectively. Substituting these into the objective and constraint proves both equivalences. □

For the two leaves, a shorter action on the first saves $c > 0$ but incurs $r > 0$ additional misses, while a longer action on the second costs $c' > 0$ and removes $r' > 0$ misses. The replacement has

$$\Delta C_J = c' - c, \qquad \Delta M_J = r - r', \qquad \Delta\,\mathrm{Recall} = \frac{r' - r}{mk}. \tag{25}$$

If $r' \geq r$ and $c' < c$, the training recall is preserved or improved while the cost strictly decreases. This is the reason why the fitted policy may prefer a shorter continuation even if the longer one recovers another neighbor. Under the training constraint, equivalent recall gain can be obtained from other queries at lower cost. The condition holds for all training queries in a leaf through their accumulated costs and misses.

C.2 Search Trajectories and Neighbor Discovery

Proof of Lemma 1. Fix the query, graph, arithmetic, and deterministic ordering of equal scores. Both executions start with the same width-64 frontier and visited and result sets. If the frontier empties during the prefix, their results are identical. Otherwise, both execute the first 32 distinct visits and resize the frontier to width w.

Assume that frontier, processing position, visited identifiers and result lists are the same before the pop operation. Both operations pop the same identifier. An already visited identifier is processed by both without changing the result list. For a new identifier, both compute the same distance and update the result list in the same way. Before the stopping budget is reached, both score the same outgoing edges and perform the same insertions. Both states are identical.

At the bth distinct visit, stopping returns the updated result list. Further insertion of outgoing edges in the full trajectory affects only the frontier. Its checkpoint result is unchanged. At $b = 32$, both complete the shared prefix before the decision. Two identical states imply identical exhaustion if the requested budget exceeds the length of the trajectory. □

Let $x^w_{i,1}, \ldots, x^w_{i,n_i(w)}$ be the distinct vertices in the full width-w trajectory, including the shared prefix. Define

$$v_i(w, b) = \min\{b, n_i(w)\}, \qquad V_i(w, b) = \{x^w_{i,t} \mid 1 \leq t \leq v_i(w, b)\}. \tag{26}$$

For $b = \infty$, we have $v_i(w, b) = n_i(w)$. The visited sets are nested for a fixed width. Changing the width may change the trajectory, and is considered a separate action family.

Lemma 3 (Discovery representation)**.** *We use the same distance ranking and its rule for ties to define $N_k(q_i)$ and to keep the closest visited vertices. If $y_{ij} \in N_k(q_i)$, we define*

$$\tau_{ij}(w) = \min\{t \mid x^w_{i,t} = y_{ij}\}, \qquad \min \varnothing = \infty. \tag{27}$$

Then, Equation (6) holds and $u_i(w, b)$ is nonincreasing in b.

Proof. A top-k vertex in the database has at most $k - 1$ predecessors in the common ranking. Once visited, it becomes a top-k vertex in every visited superset. A vertex that has not been visited cannot be in the result. Hence,

$$S(q_i, w, b) \cap N_k(q_i) = V_i(w, b) \cap N_k(q_i). \tag{28}$$

Reference neighbor j is missing exactly when $\tau_{ij}(w) > v_i(w, b)$. Summing these indicators yields Equation (6). Increasing b increases the finite visited prefix, so none of the indicators increase. □

For budgets $b_1 < b_2$, the improvement is

$$u_i(w, b_1) - u_i(w, b_2) = \sum_{j=1}^{k} \mathbf{1}\{v_i(w, b_1) < \tau_{ij}(w) \leq v_i(w, b_2)\}. \tag{29}$$

The recovery is constant between discovery visits. The next improvement requires traversing the intermediate portion of the graph, whose length is not necessarily similar for different queries. Equation (4) compares these improvements by the number of recovered neighbors and the measured costs.

Collection cost. Let $\mathcal{B}_w$ be the budget checkpoints for width w, including a checkpoint at exhaustion. Collecting one full trajectory per width requires

$$N_{\text{path}} = \sum_i \sum_{w \in \mathcal{W}} n_i(w) \tag{30}$$

full-vector distance evaluations per collection pass. Computing each checkpoint independently would require

$$N_{\text{separate}} = \sum_i \sum_{w \in \mathcal{W}} \sum_{b \in \mathcal{B}_w} \min\{b, n_i(w)\}, \qquad 1 \le \frac{N_{\text{separate}}}{N_{\text{path}}} \le \max_w |\mathcal{B}_w|. \tag{31}$$

For one trajectory of length $n \ge h \ge 1$ and every integer checkpoint from h to n, the ratio is $(n-h+1)(n+h)/(2n)$. Checkpoint labeling requires $O(k^2)$ identifier comparisons using the direct comparison method, but no distance evaluations. The collection routine records these costs and subtracts label-comparison time from elapsed trajectory measurements.

C.3 Partition Learning and Shared Actions

Fix a price $\lambda \ge 0$ and a finite common action set. Define $z_i(a) = t_i(a) + \lambda u_i(a)$, and

$$\mathcal{O}_\lambda(I) = \arg\min_{a \in \mathcal{A}} \sum_{i \in I} z_i(a), \qquad \mathcal{R}_\lambda(\mathcal{P}) = \sum_{I \in \mathcal{P}} H_\lambda(I) - \sum_i \min_{a \in \mathcal{A}} z_i(a), \tag{32}$$

where the sum is over active queries. The nonnegative number $\mathcal{R}_\lambda(\mathcal{P})$ is the increase in penalized cost incurred by forcing all queries in each leaf to share an action.

Proposition 3 (Split gain)**.** *Equation (8) is nonnegative for disjoint nonempty children I^- and I^+ with union I. It is zero if and only if*

$$\mathcal{O}_\lambda(I^-) \cap \mathcal{O}_\lambda(I^+) \ne \varnothing. \tag{33}$$

Replacing I by its children decreases $\mathcal{R}_\lambda$ by $G_\lambda(I^-, I^+)$.

Proof. For each action a, each child's cost is at least its own minimum. Summing these inequalities and minimizing over a yields the nonnegativity result. A common minimizer is one that attains both child minima and their sum, resulting in zero gain. Conversely, suppose a^* is a minimizer of the parent cost. If the gain is zero, then

$$\left(\sum_{i \in I^-} z_i(a^*) - H_\lambda(I^-)\right) + \left(\sum_{i \in I^+} z_i(a^*) - H_\lambda(I^+)\right) = 0. \tag{34}$$

Each term is nonnegative, so they are both zero and a^* minimizes each child. In Equation (32), a split only affects the parent term, so the last claim holds. □

For any sequence of splits from $\mathcal{P}$ to $\mathcal{P}'$, we have the telescoping identity

$$\mathcal{R}_\lambda(\mathcal{P}) - \mathcal{R}_\lambda(\mathcal{P}') = \sum_{\text{splits } s} G_\lambda(s). \tag{35}$$

This equality links the feature thresholds with the continuation decisions. A split is beneficial if the children prefer different actions, including where the current retrieval quality is similar. In dense refinement, the candidates are from the parent and the minima of per-query loss. The parent minimizer is included, so any accepted split retains the option of giving both children the same action as the parent.

Partition regret can also be bounded using within-leaf variation. Fix a representative query $r(I) \in I$ and define

$$\varepsilon_i = \max_{a \in \mathcal{A}} |z_i(a) - z_{r(I)}(a)|. \tag{36}$$

If a_r minimizes $z_{r(I)}$ and a_i minimizes z_i, then

$$z_i(a_r) - z_i(a_i) \le z_{r(I)}(a_r) - z_{r(I)}(a_i) + 2\varepsilon_i \le 2\varepsilon_i. \tag{37}$$

The best shared action costs no more than a_r, so

$$0 \le \mathcal{R}_\lambda(\mathcal{P}) \le 2 \sum_i \varepsilon_i. \tag{38}$$

Queries with similar penalized action costs can share a continuation with a correspondingly small increase in the fitting objective.

C.4 Optimal Allocation, Dominance, and Refinement

Proof of Proposition 1. For zero leaves, the only feasible miss count is zero with cost zero. Let $F_{\ell-1}(s)$ be the minimum cost of getting s misses on the first $\ell-1$ leaves. An assignment of ℓ leaves with s misses has an action a on leaf ℓ. The leaves preceding it have $s - M_\ell(a)$ misses and cost at least $F_{\ell-1}(s - M_\ell(a))$. The minimum is (11).

Conversely, every finite term in this recurrence is associated with an attainable predecessor state and one action on the new leaf. It achieves the desired cost and miss count. Induction proves optimality for every terminal state. Minimizing over $s \le B - M_0$, adding C_0, and backtracking gives Equation (10) and its assignment.

We now split a leaf into children with the same set of available actions and per-query arrays. We assign each child the parent's action, which keeps both sums in Equation (9) the same. Each old feasible assignment is available with the same cost. Minimizing over the refined partition cannot increase the optimum. Repeating this argument proves Equation (12). □

Dominance. If $C_\ell(a') \le C_\ell(a)$ and $M_\ell(a') \le M_\ell(a)$, then replacing a with a' preserves feasibility and does not increase cost. Equal-miss actions can be replaced by their lowest-cost representative. In the remaining nondominated set, increasing miss count strictly decreases cost. Sorting by misses and keeping only new cost minima defines this set in $O(|\mathcal{A}_\ell| \log |\mathcal{A}_\ell|)$ time.

Penalty selection. Penalized choices may exclude points on the discrete cost-recall curve. If the miss-cost pairs are $(0, 10)$, $(1, 6)$, and $(2, 0)$, then minimizing $C + \lambda M$ to choose the middle point mandates

$$6 + \lambda \le 10 \quad \text{and} \quad 6 + \lambda \le 2\lambda, \tag{39}$$

which requires $\lambda \le 4$ and $\lambda \ge 6$. No price can accommodate both. Nonetheless, with budget one, this is the point with minimum feasible cost. Integer miss allocation retains such choices. For $\lambda_2 > \lambda_1$, any price minimizers must satisfy

$$(\lambda_2 - \lambda_1)\big(M_\ell(a_{\lambda_2}) - M_\ell(a_{\lambda_1})\big) \le 0, \tag{40}$$

from adding the two optimality inequalities. Increasing the price reduces misses via discrete jumps. Dynamic programming finds this optimum directly over their possible sums.

C.5 Dual Bounds and Discrete Action Costs

For $\lambda \ge 0$, the Lagrangian is

$$\mathcal{L}(\boldsymbol{a}, \lambda) = C(\boldsymbol{a}) + \lambda\big(M(\boldsymbol{a}) - B\big) \tag{41}$$

$$= C_0 + \lambda(M_0 - B) + \sum_{\ell=1}^{L} \big(C_\ell(a_\ell) + \lambda M_\ell(a_\ell)\big). \tag{42}$$

Minimizing over the Cartesian product of the leaf actions, we obtain $D_{\mathcal{P}}(\lambda)$ as in Equation (13). For every feasible assignment, $\mathcal{L}(\boldsymbol{a}, \lambda) \le C(\boldsymbol{a})$, so we have a lower bound by Lagrangian duality (Boyd & Vandenberghe, 2004).

The nonnegative reduced cost of an action is defined by

$$\delta_\ell(a, \lambda) = C_\ell(a) + \lambda M_\ell(a) - \min_{a' \in \mathcal{A}_\ell} \{C_\ell(a') + \lambda M_\ell(a')\}. \tag{43}$$

For any feasible $\boldsymbol{a}$ this gives

$$C(\boldsymbol{a}) - D_{\mathcal{P}}(\lambda) = \sum_{\ell=1}^{L} \delta_\ell(a_\ell, \lambda) + \lambda\big(B - M(\boldsymbol{a})\big), \tag{44}$$

$$0 \le C(\boldsymbol{a}) - V_{\mathcal{P}}(B) \le \sum_{\ell=1}^{L} \delta_\ell(a_\ell, \lambda) + \lambda\big(B - M(\boldsymbol{a})\big). \tag{45}$$

where the right hand side accounts for deviations from individually priced actions and unused miss budget. This vanishes when every leaf minimizes its priced loss and $\lambda(B - M(\boldsymbol{a})) = 0$.

Proof of Proposition 2. Introduce weights $p_{\ell a} \geq 0$ such that $\sum_a p_{\ell a} = 1$ in each leaf. Relaxing the requirement of a single action yields the finite linear program

$$\begin{aligned} V_{\mathrm{LP}}(B) = \min_{p} \quad & C_0 + \sum_{\ell,a} p_{\ell a} C_\ell(a) \\ \text{subject to} \quad & M_0 + \sum_{\ell,a} p_{\ell a} M_\ell(a) \leq B, \\ & \sum_a p_{\ell a} = 1 \quad \text{for each } \ell. \end{aligned} \tag{46}$$

These weights represent convex combinations of measured action outcomes. Minimizing the Lagrangian over each simplex yields the minimum of its action costs, so this program has dual $\max_{\lambda \geq 0} D_{\mathcal{P}}(\lambda)$. Feasibility and finite action costs imply a bounded optimum. Linear-programming duality gives equality between the two values.

Choose an optimal extreme point. If its budget inequality is active, the positive variables have at most $L + 1$ independent affine constraints, which include L simplex constraints and the budget inequality. More than $L + 1$ positive variables would have a nonzero perturbation that still satisfied all of these constraints. Small enough perturbations of either sign would preserve nonnegativity, which contradicts extremality. Each leaf must have at least one positive variable. So at most one leaf can have two actions and all others will have one. If the budget inequality is slack, then the same argument using the L simplex constraints gives one action in each leaf.

Consider leaf j that mixes a^- and a^+, ordered so $M_j(a^-) \leq M_j(a^+)$. Let its mixture be $\theta a^- + (1-\theta)a^+$ with $0 < \theta < 1$. Replacing it by a^- cannot increase the total number of misses. The increase in its cost is

$$C_j(a^-) - \big(\theta C_j(a^-) + (1-\theta) C_j(a^+)\big) = (1-\theta)\big(C_j(a^-) - C_j(a^+)\big) \leq \Delta_j. \tag{47}$$

This yields a feasible single-action assignment with cost at most $V_{\mathrm{LP}}(B) + \Delta_j$. The integer optimum cannot be larger than this assignment, and cannot be smaller than its relaxation. Thus (14) holds. □

Leaf size. If $\max_a t_i(a) - \min_a t_i(a) \leq H$ for every active query, we have

$$\Delta_\ell \leq \sum_{i \in I_\ell} \big(\max_a t_i(a) - \min_a t_i(a)\big) \leq |I_\ell| H. \tag{48}$$

and then dividing (14) by m gives

$$\frac{V_{\mathcal{P}}(B) - \max_{\lambda \geq 0} D_{\mathcal{P}}(\lambda)}{m} \leq H \frac{\max_\ell |I_\ell|}{m}. \tag{49}$$

A smaller maximum leaf size tightens the bound for the discrete action costs compared with the fractional relaxation. Furthermore, Equation (12) ensures that refinement maintains all allocations available before.

C.6 Recall Constraints and Marginal Costs

Given a fixed partition, the minimum miss budget is

$$B_{\min} = M_0 + \sum_{\ell=1}^{L} \min_{a \in \mathcal{A}_\ell} M_\ell(a). \tag{50}$$

An allocation is feasible if and only if $B \geq B_{\min}$; necessity is shown by summing the leafwise minima, sufficiency by choosing a minimum-miss action in each leaf. Feasible sets are nested with increasing B, so $V_{\mathcal{P}}(B)$ is nonincreasing. Integer miss counts imply

$$V_{\mathcal{P}}(B) = V_{\mathcal{P}}(\lfloor B \rfloor). \tag{51}$$

In each leaf, let $\bar{a}_\ell$ be a minimum-cost action, breaking ties by smaller miss count. If $B \geq B_{\mathrm{free}} = M_0 + \sum_\ell M_\ell(\bar{a}_\ell)$, then

$$V_{\mathcal{P}}(B) = C_0 + \sum_\ell C_\ell(\bar{a}_\ell). \tag{52}$$

Further relaxing recall cannot improve this value, since each leaf already uses its minimum-cost action.

Continuous value function. Let us allow B to take real values in the linear program of Equation (46) and call its value $W_{\mathcal{P}}(B)$. The Lagrangian representation of the value function is

$$W_{\mathcal{P}}(B) = \max_{\lambda \geq 0} \{K(\lambda) - \lambda B\}, \qquad K(\lambda) = C_0 + \lambda M_0 + \sum_{\ell} \min_{a} \{C_\ell(a) + \lambda M_\ell(a)\}. \tag{53}$$

It is convex, nonincreasing, and piecewise linear on $[B_{\min}, \infty)$. Convexity and monotonicity follow from the supremum of affine functions with nonpositive slopes. Piecewise linearity follows from the finiteness of the action sets and their polyhedral feasible region.

When choosing an optimal multiplier λ_B, applying the same multiplier at any other feasible budget B' gives

$$W_{\mathcal{P}}(B') \geq W_{\mathcal{P}}(B) - \lambda_B(B' - B). \tag{54}$$

Applying this inequality in both directions gives Equation (15). In particular, $\lambda_1 \geq \lambda_2$ for $B_1 < B_2$. At a differentiable interior point, $W'_{\mathcal{P}}(B) = -\lambda_B$. Let ρ be a continuous recall target. Let the per-query cost be

$$g(\rho) = \frac{W_{\mathcal{P}}(mk(1-\rho))}{m}. \tag{55}$$

The chain rule gives

$$g'(\rho) = k\lambda_{mk(1-\rho)}. \tag{56}$$

The multiplier represents the computational cost of an additional recovered neighbor. Increasing recall becomes progressively more expensive.

Let $\Delta_{\mathcal{P}} = \max_\ell \Delta_\ell$. By combining Proposition 2 and Equation (15), we obtain for feasible $B_1 < B_2$,

$$\max\{0, \lambda_2(B_2 - B_1) - \Delta_{\mathcal{P}}\} \leq V_{\mathcal{P}}(B_1) - V_{\mathcal{P}}(B_2) \leq \lambda_1(B_2 - B_1) + \Delta_{\mathcal{P}}. \tag{57}$$

This follows since each integer optimum is within $[0, \Delta_{\mathcal{P}}]$ of the corresponding fractional optimum, and the difference of the two errors is at most $\Delta_{\mathcal{P}}$ in absolute value.

Allocation along action frontiers. The continuous optimum can also be constructed from the leafwise cost-recovery curves. Keep only nondominated actions and recover relative to $\bar{a}_\ell$,

$$r_\ell(a) = M_\ell(\bar{a}_\ell) - M_\ell(a), \qquad c_\ell(a) = C_\ell(a) - C_\ell(\bar{a}_\ell). \tag{58}$$

Both coordinates are nonnegative. Let $(r_{\ell,j}, c_{\ell,j})$, $j = 0, \ldots, s_\ell$, be the vertices of the lower convex hull, sorted by increasing recovery from $(0, 0)$. The lengths and slopes of consecutive segments are

$$d_{\ell,j} = r_{\ell,j} - r_{\ell,j-1} > 0, \qquad \gamma_{\ell,j} = \frac{c_{\ell,j} - c_{\ell,j-1}}{d_{\ell,j}}, \qquad \gamma_{\ell,j+1} \geq \gamma_{\ell,j} > 0. \tag{59}$$

For a budget $B < B_{\text{free}}$, we need to recover $R = B_{\text{free}} - B$ in total relative to cheapest actions. Let $y_{\ell,j}$ be the recovery obtained along a segment. The additional fractional cost is the value of

$$\begin{aligned} \min_{y} \quad & \sum_{\ell,j} \gamma_{\ell,j} y_{\ell,j} \\ \text{subject to} \quad & \sum_{\ell,j} y_{\ell,j} \geq R, \qquad 0 \leq y_{\ell,j} \leq d_{\ell,j}. \end{aligned} \tag{60}$$

The segments of each leaf are used in hull order. Nondecreasing slopes ensure that a global ordering by $\gamma_{\ell,j}$ respects this order, with ties broken by hull order within each leaf.

Proposition 4 (Marginal recovery allocation)**.** *For $0 < R \leq B_{\text{free}} - B_{\min}$, filling the segments of Equation (60) in nondecreasing slope order until the total recovery reaches R gives an assignment of total cost $W_{\mathcal{P}}(B)$. At most one segment is partially filled.*

Proof. Take any feasible vector y with positive mass on a segment of slope γ', and unutilized capacity on a segment of smaller slope γ. Transferring an amount ϵ from the former to the latter segment maintains the total recovery and changes the cost by $\epsilon(\gamma - \gamma') < 0$. Equal-slope transfers leave cost unchanged. Iterating these transfers fills up the cheaper segments first. Since all slopes are positive, an optimum must use total recovery R. It can have at most one segment partially filled, by consistent tie breaking.

This filling traces out the lower convex hull of each leaf. Each traced point is a convex combination of two adjacent actions, and is thus feasible in Equation (46). On the other hand, every feasible fractional assignment projects to a point on or above its lower hull. The cost of the point is at least the segment solution for the same recovery. The two optima are the same after adding $C_0 + \sum_\ell C_\ell(\bar{a}_\ell)$. □

If the last segment is partial, the slope of the last segment is an optimal multiplier. At a boundary of two segments, any nonnegative price between the slope of the last selected segment and the slope of the first unselected segment can support the allocation. The integer problem can make use of nondominated actions above the lower hull, as the example in Appendix C.4 with three actions shows. Dynamic programming chooses among all such actions under the integer budget.

C.7 Sensitivity to Measured Costs

Proposition 5 (Cost perturbations)**.** *Let $t_i(a)$ and $\widetilde{t}_i(a)$ be two cost functions on the same feasible assignments with the same fixed cost C_0, such that*

$$\max_a |t_i(a) - \widetilde{t}_i(a)| \le \epsilon_i, \qquad E = \sum_i \epsilon_i. \tag{61}$$

If $\boldsymbol{a}^*$ minimizes the first cost and $\widetilde{\boldsymbol{a}}$ is within $\delta \ge 0$ of the minimum of the second, then

$$C(\widetilde{\boldsymbol{a}}) \le C(\boldsymbol{a}^*) + 2E + \delta. \tag{62}$$

Proof. Every assignment changes cost by at most E, so

$$C(\widetilde{\boldsymbol{a}}) \le \widetilde{C}(\widetilde{\boldsymbol{a}}) + E \le \widetilde{C}(\boldsymbol{a}^*) + \delta + E \le C(\boldsymbol{a}^*) + \delta + 2E. \tag{63}$$

The feasible set, which depends only on the miss counts, is unaffected by the cost perturbation. □

Given a multiplicative perturbation $(1-\epsilon)t_i(a) \le \widetilde{t}_i(a) \le (1+\epsilon)t_i(a)$, for any $0 \le \epsilon < 1$, the optimal assignment for $\widetilde{t}$ satisfies:

$$C(\widetilde{\boldsymbol{a}}) \le \frac{1+\epsilon}{1-\epsilon} C(\boldsymbol{a}^*). \tag{64}$$

This follows from $(1-\epsilon)C(\widetilde{\boldsymbol{a}}) \le \widetilde{C}(\widetilde{\boldsymbol{a}}) \le \widetilde{C}(\boldsymbol{a}^*) \le (1+\epsilon)C(\boldsymbol{a}^*)$. These bounds quantify how changes in the measured action costs affect the optimal allocation given a fixed recall constraint.

C.8 Geometry of Search Features

Let $z_j = d_j^{(32)} > 0$ and $s_j = r(x_j^{(32)}) > 0$ be distances and scales before numerical floors. For the transformations $x' = cUx + t$ and $q' = cUq + t$, with $c > 0$ and $U^\top U = I$, the corresponding distances satisfy $z'_j = c^2 z_j$ and $s'_j = c^2 s_j$. This gives

$$\frac{z'_j}{z'_{10}} = \frac{z_j}{z_{10}}, \qquad \frac{z'_{10}}{\bar{s}'} = \frac{z_{10}}{\bar{s}}, \qquad \frac{\sigma_{z'}}{\bar{z}'} = \frac{\sigma_z}{\bar{z}}, \qquad \log z'_j = \log z_j + 2\log c. \tag{65}$$

Relative distances, local-scale ratios and coefficients of variation are unchanged in corresponding search states. Logarithmic distances carry absolute scale information. Progress ratios at different visits have the same cancellation when the histories match.

Distance concentration features have a local likelihood interpretation (as in nearest neighbor dimension estimation) (Levina & Bickel, 2004). Suppose we have a reference model with probability mass within Euclidean radius r proportional to r^s. Then for squared distance Z, we have mass proportional to Z^α, where $\alpha = s/2$. Conditional on $Z \le z_r$, the normalized radius $U = Z/z_r$ has density

$$f_\alpha(u) = \alpha u^{\alpha-1}, \qquad 0 < u < 1. \tag{66}$$

For $r-1$ independent samples within this radius, the log likelihood (up to a constant independent of α) is

$$\ell(\alpha) = (r-1)\log\alpha + (\alpha-1)\sum_{j=1}^{r-1} \log\frac{z_j}{z_r}. \tag{67}$$

Taking the derivative, we have

$$\ell'(\alpha) = \frac{r-1}{\alpha} + \sum_{j=1}^{r-1} \log \frac{z_j}{z_r}, \qquad \ell''(\alpha) = -\frac{r-1}{\alpha^2} < 0, \tag{68}$$

and the maximum likelihood estimate is

$$\widehat{\alpha}_r = -\left(\frac{1}{r-1} \sum_{j=1}^{r-1} \log \frac{z_j}{z_r} \right)^{-1}. \tag{69}$$

Features 7 and 8 compute the clipped versions of this statistic for $r = 16$ and $r = 10$ on the visited distances. Squared distances have exponent half the local dimension of the reference model. The feature measures the concentration of the neighborhood already visited by search.

Distribution of the concentration statistic. For the unclipped statistic in the radial model, let $\nu = r - 1$, $Y_j = -\log(z_j/z_r)$, and $T = \sum_{j=1}^{\nu} Y_j$. The transformed density f_α satisfies

$$f_Y(y) = \alpha e^{-\alpha y}, \qquad y > 0, \qquad f_T(t) = \frac{\alpha^\nu}{\Gamma(\nu)} t^{\nu-1} e^{-\alpha t}. \tag{70}$$

Ordering the interior distances makes no difference to the log-sum or its distribution. Since $\widehat{\alpha}_r = \nu/T$, integration against this gamma density shows that for $0 < p < \nu$

$$\mathbb{E}[T^{-p}] = \alpha^p \frac{\Gamma(\nu - p)}{\Gamma(\nu)}. \tag{71}$$

For $\nu > 2$, its first two moments are

$$\mathbb{E}[\widehat{\alpha}_r] = \frac{\nu\alpha}{\nu - 1}, \qquad \mathrm{Var}(\widehat{\alpha}_r) = \frac{\nu^2 \alpha^2}{(\nu-1)^2(\nu-2)}. \tag{72}$$

Its coefficient of variation is $1/\sqrt{\nu - 2}$. The statistic is stabilized in this model with additional interior distances. The two feature coordinates have different neighborhood sizes, giving the policy an estimate of concentration at both scales.

C.9 Traversal Work and Policy Storage

Let v_i, E_i, J_i, and P_i be the numbers of distinct visits, scored edges, frontier insertions, and frontier pops, respectively. For queries ending in the prefix, let $w_i = 64$ and $b_i = \infty$. If the graph degree is at most Γ,

$$v_i \le \min\{b_i, n_i(w_i)\}, \qquad E_i \le \Gamma v_i, \qquad J_i \le E_i, \qquad P_i \le J_i + 1. \tag{73}$$

The final inequality includes the initial entry vertex. Duplicate pops do not calculate another full-vector distance. Edges at the final budgeted visit can be omitted after the result list is updated.

Let $s(d)$ be the cost of one quantized edge estimate, D_π be the tree depth, and $T_{\mathrm{init}}(n, d)$ be the initialization cost. The sorted array frontier performs a binary search followed by a shift of at most w_i entries for each insertion. A pop operation can advance past at most w_i entries that have been processed. Given a result list of size $O(k)$, we have the upper bound

$$T_i = O(T_{\mathrm{init}}(n, d) + v_i(d + k) + s(d)E_i + w_i(J_i + P_i) + D_\pi)\,. \tag{74}$$

Feature computation involves $O(d + k)$ arithmetic operations at the fixed prefix. The full vector part alone involves $\Theta(dv_i)$ work. A reduction in visits saves distance work and reduces the number of edge scores. The width also impacts the cost of maintaining alternative candidates. This motivates the joint selection of these parameters.

A full binary tree with L leaves contains $L - 1$ internal nodes. Each stored node contains four 32-bit integers and one FP32 threshold. Its node payload and local-scale storage are

$$S_{\mathrm{tree}} = 20(2L - 1) \text{ bytes}, \qquad S_{\mathrm{scale}} = 4n \text{ bytes}. \tag{75}$$

Inference traverses at most $D_\pi \le 14$ internal nodes per active query. Workers share the immutable payload and maintain separate frontiers, result lists, and visited sets.

Parallel query scheduling. There is a set of independent queries available at time zero, with fixed processing times $\tau_1, \ldots, \tau_m$ on p identical workers. Let $S = \sum_i \tau_i$ and $\tau_{\max} = \max_i \tau_i$. The next available query is assigned to the first idle worker: this is list scheduling (Graham, 1966) with completion time T_p satisfying

$$\max\{S/p, \tau_{\max}\} \le T_p \le S/p + (1 - 1/p)\tau_{\max}. \tag{76}$$

The lower bounds are obtained from the total work and the longest query. For the upper bound, let query j be the last to finish, and let s_j be its start time. Before s_j, all workers are occupied whenever query j is waiting, so $ps_j \le S - \tau_j$. Therefore

$$T_p = s_j + \tau_j \le (S - \tau_j)/p + \tau_j \le S/p + (1 - 1/p)\tau_{\max}. \tag{77}$$

When $S > 0$, the corresponding speedup is

$$\frac{p}{1 + (p-1)\tau_{\max}/S} \le \frac{S}{T_p} \le p. \tag{78}$$

The implementation assigns one query per dynamic scheduling step. Total search cost limits the average work per worker, while the longest query sets the additive scheduling term.

C.10 Byte Distances for SIFT Descriptors

For SIFT1M, both source and query coordinates are integers in $[0, 255]$. Let $z_j = x_j - 128$, and store them as signed bytes. The expanded form of squared Euclidean distance is

$$\|q - x\|_2^2 = \sum_{j=1}^{128} q_j^2 + \sum_{j=1}^{128} x_j^2 - 2\sum_{j=1}^{128} q_j x_j \tag{79}$$

$$= \|q\|_2^2 + \|x\|_2^2 - 2\left(\sum_{j=1}^{128} q_j z_j + 128\sum_{j=1}^{128} q_j\right). \tag{80}$$

AVX-512 VNNI computes the unsigned-signed byte products with integer accumulation. The source norms are stored, and the query norms and sums are computed once. The signed dot product is bounded as

$$-128 \cdot 255 \cdot 128 \le \sum_{j=1}^{128} q_j z_j \le 128 \cdot 255 \cdot 127, \tag{81}$$

which is within the signed 32-bit range. The norm sums and doubled unshifted inner products are at most $2 \cdot 128 \cdot 255^2 = 16{,}646{,}400$, also within the range. The squared distance is bounded as

$$0 \le \|q - x\|_2^2 \le 128 \cdot 255^2 = 8{,}323{,}200 < 2^{24}. \tag{82}$$

and all possible integer values are representable in FP32. The descriptor geometry is preserved by the conversion and distance computation.

D Experimental Setup

D.1 Dataset Preparation

SIFT1M, GIST1M, and GloVe100 use the public vector files distributed through ANN-Benchmarks (Aumüller et al., 2020). DEEP-1M uses the 256-dimensional million vector collection distributed with the quantization and graph evaluations (Gao & Long, 2024; Gou et al., 2025). Database rows keep their identifiers and ordering.

SIFT1M, DEEP-1M, and GIST1M use squared Euclidean distance on the coordinates they provide. GloVe100 database and query rows are divided by their Euclidean norms in FP64 and stored as FP32. Reference neighbors are computed on the normalized FP32 rows. Training and test queries are preprocessed identically and searched on the same database. Training queries provide the trajectory measurements for policy fitting, while the separate test sets in Table 1 provide the reported recall, throughput, and visit statistics. All methods receive the same database and test matrices. Recall@10 counts intersections with the 10 reference identifiers, including the identifiers selected in the reference results when distances tie.

D.2 Software and Hardware

Experiments are run on two Intel Xeon Platinum 8352V sockets, each with 32 physical cores, two hardware threads per core, and a nominal base frequency of 2.10 GHz. Search is pinned to CPUs 32 for one core and CPUs 32–39 for eight cores on NUMA node 1. Software versions are Linux 5.15, glibc 2.35, Python 3.12.3, and NumPy 2.3.2. Native compilation of TraCANN uses C++17, `-O3`, `-march=native`, and OpenMP.

Table 6: Native implementations used in the comparison.

Method	Implementation	Version
HNSW	`hnswlib`	0.8.0
SymphonyQG	`rabitqlib` quantized graph	0.3.6
SVS/LVQ	`scalable-vs`	0.4.0
LoRANN	`lorann`	1.0
RaBitQ-IVF	`rabitqlib` IVF	0.3.6
ScaNN	`scann`	1.4.2
OPQ+FastScan	`faiss-cpu`	1.15.1
HNSW+KS2	Authors' KS implementation	Public source
AdaptNN	Authors' Faiss fork	1.5.2
DARTH	Authors' Faiss fork	1.8.0
DABS	Authors' HNSW implementation	Public source

Baselines use their public construction and query interfaces. SymphonyQG uses the maintained RaBitQ-Library graph implementation with PiPNN initialization. RaBitQ-IVF includes the library's multi-bit quantized modes and its 32-bit raw-vector mode. AdaptNN and DARTH use LightGBM 2.3.1 and 4.5.0 respectively. Array interfaces call the authors' search and distance kernels. AdaptNN uses the authors' SSE4/POPCNT build with OpenMP, and DABS uses an optimized AVX build.

D.3 Direct Construction and Query Configurations

Direct configurations use official examples and API defaults, and required parameters are specified below. Tuples are in the order of SIFT1M, DEEP-1M, GIST1M, and GloVe100.

- HNSW is used with $M = 16$, 200 construction effort, random seed 100, and query effort of 50 from the authors' example.
- SymphonyQG uses degree 32, construction effort 200, PiPNN initialization and raw stored vectors. Query effort is 100.
- SVS/LVQ has 4 primary bits and 8 residual bits, graph degree 64, construction window 128, $\alpha = 1.2$, zero loader padding, and query window 20.
- LoRANN uses 1,024 clusters and rank 32. SIFT1M follows the official example with 4-bit quantization and no global dimension reduction. Other datasets use 8-bit quantization, with global dimension 128 when the input dimension is larger than 128, as in the documented cluster-size and dimension setting. Queries search 32 clusters and rerank 800 points.
- RaBitQ-IVF uses 256 clusters, 8-bit quantization, and native IVF training on the full database. Faster quantization is disabled (as in the example's default settings). The seven probe counts $\{5, 10, 20, 40, 80, 120, 200\}$ define the direct curve.
- ScaNN uses the public automatic builder with mode `NONE`, FP32 reranking, and the native query defaults.
- OPQ+FastScan. The index employs IVF4096, 4-bit PQ, and full-vector reranking, where the subquantizer counts for PQ are $(32, 64, 120, 25)$. OPQ and IVF training are run on a sample of 200,000 rows, with a seed of 42, and native iteration counts. Direct search probes one list and the reranking multiplier is set to one.
- HNSW+KS2 uses $M = 32$, construction effort $(1000, 1000, 1000, 2000)$, and kernel partition counts $(8, 16, 20, 5)$. SIFT1M and GIST1M follow the authors' build settings. DEEP-1M and GloVe100 use the dimension-based partition rule. The direct query curve uses effort 10 through 100 in steps of 10. The native query implementation is serial.

- AdaptNN employs HNSW with $M = 16$ and construction effort 40. The gradient-boosted regressors employ 100 trees, 31 leaves, and learning rate 0.2, with query coordinates and native search-state features. Training predicts the base-two logarithm of the number of distance evaluations required to find the nearest neighbor. Direct search employs observation thresholds $(241, 368, 1260, 512)$, and the authors' multiplier lists for SIFT, DEEP, and GIST. SIFT10M and DEEP10M configurations are applied to the million-vector datasets. GloVe100 uses threshold 512 and multiplier 100 as an extension of these configurations to the dataset.
- DARTH uses HNSW degrees $(32, 32, 32, 16)$ and construction effort 500. Training trajectories use search efforts $(500, 750, 1000, 500)$ and logging interval 2. Its regressor has 100 trees, 31 leaves, learning rate 0.1, seed 42, and eleven search-state features in the order consumed by the native predictor. Direct search requests recall in $\{0.95, 0.99, 0.995\}$ with initial prediction interval 1000, minimum interval 100, and the dataset's trajectory search effort. SIFT and DEEP use the authors' larger dataset construction settings.
- DABS uses degrees $(14, 14, 24, 16)$ for HNSW, construction effort 500, and keeps the original insertion order and seed. Distance termination compares the nearest frontier candidate to $(1 + \gamma)$ times the current 10th neighbor distance, using Euclidean distances in the stopping rule. Direct search uses $\gamma = 0.4$. GloVe100 uses the degree specified for GloVe in the paper.

AdaptNN's direct multipliers: $\{91, 149, 248, 354, 399, 463, 539, 729, 3816\}$ for SIFT1M, $\{135, 257, 402, 471, 576, 736, 1115, 7259\}$ for DEEP-1M, and $\{317, 478, 548, 630, 772, 1282, 14899\}$ for GIST1M. The multipliers are taken from the authors' example curves.

D.4 Construction Variants and Query Parameter Search

Additional construction configurations are included in parameter tuning. HNSW uses $M = 32$ with construction effort 400. SymphonyQG uses degree 32 with construction effort 400 and degree 64 with construction effort 800. SVS/LVQ uses construction window 200 and padding 32. LoRANN uses 2,048 clusters, global dimension 192 for DEEP-1M and GIST1M, and 4-bit quantization for GloVe100. RaBitQ-IVF uses 4,096 clusters, 7-bit and 32-bit storage, a 200,000-row training sample, 20 clustering iterations, and seed 42. Its raw-vector mode uses the same clustering. ScaNN uses a 1,024-leaf tree, 2-dimensional anisotropic-quantization blocks, and a 100,000-row training sample. OPQ+FastScan uses 240 PQ subquantizers on GIST1M. AdaptNN, DARTH, and DABS preserve their construction settings during query tuning. Direct and tuned results use 80 construction configurations across 4 datasets and 11 methods.

For HNSW, SymphonyQG, and KS2, the grid of graph query effort values is

$$\{10, 20, 30, 40, 60, 80, 100, 140, 200, 300, 450, 650, 1000, 1500, 2500, 4000\}. \tag{83}$$

For SVS, the same values are used up to 2500 for its search window. RaBitQ-IVF probe counts are

$$\{4, 8, 16, 32, 64, 96, 128, 192, 256, 384, 512, 768, 1024, 2048, 4096\}, \tag{84}$$

clipped to the number of index lists. LoRANN varies searched clusters over $\{8, 16, 32, 64, 128, 256\}$ and reranking counts over $\{100, 200, 400, 800, 1600, 3200\}$. ScaNN uses the same reranking grid and searched leaves $\{8, 16, 32, 64, 128, 256, 512\}$. OPQ+FastScan varies probes over powers of two from 4 through 1024 and reranking multipliers over $\{1, 4, 16, 64, 256\}$. Multi-parameter grids use Cartesian products.

AdaptNN has observation thresholds $\{179, 241, 335, 481, 817\}$ on SIFT1M, $\{191, 265, 368, 547, 1003\}$ on DEEP-1M, $\{381, 554, 801, 1260, 2441\}$ on GIST1M, and $\{128, 256, 512, 1024, 2048\}$ on GloVe100. Budget multipliers are $\{50, 100, 150, 250, 400, 650, 1000, 1600, 2500, 4000, 8000, 16000\}$.

The requested recall for DARTH is varied over $\{0.95, 0.98, 0.99, 0.995, 0.999, 1.0\}$, the initial prediction interval over $\{128, 512, 2048\}$, and the amount of search effort over the dataset default and 2000. The minimum interval is set to one-fifth of the initial interval, rounded down. For DABS, γ is varied over $\{0, 0.001, 0.002, 0.005, 0.01, 0.02, 0.04, 0.08, 0.12, 0.2, 0.3, 0.4, 0.6\}$. Model hyperparameters are fixed in these search experiments.

Parameter trials use a full-query warmup and three timings. Increasing-effort sweeps are terminated at 99.7% recall, with up to two midpoint refinements per target. Selected candidates are formally confirmed, and report entries use the fastest confirmed setting that is eligible. Eight-core ScaNN varies its native parallel sub-batch size parameter over $\{16, 32, 64, 128\}$, and its external batch size is fixed at 128. Direct search uses its native sub-batch default value of 256. Eight-core DABS on GIST1M and GloVe100 uses candidate settings selected by the single-core sweep. GloVe100 DABS candidate settings are selected from 14 completed parameter trials, including local refinements.

HNSW, SymphonyQG, SVS, and RaBitQ methods use different direct builds with their primary query sweeps using the higher effort construction configurations above. HNSW query sweeps also used its direct builds. All other methods performed their sweeps on the specified construction variants. Maximum recall 93.75% was achieved by LoRANN on GloVe100, 97.233% using OPQ+FastScan on GloVe100, and 99.07% using KS2 on DEEP-1M. LoRANN's maximum recall GloVe100 configuration with five timing repetitions achieved 93.283%.

Confirmed recall for AdaptNN and DABS on GIST1M is at most 97.960% and 98.430% respectively. For GloVe100, the maxima are 95.742% for AdaptNN, 97.283% for DARTH, and 99.158% for DABS. Tables denote thresholds above the confirmed recall range as unreached.

D.5 Timing and CPU Placement

Each native index is loaded in a new query process on NUMA node 1 before search on CPU 32 or CPUs 32-39. Formal query configurations are measured one at a time on these cores. Concurrent construction and fitting jobs use the other CPU socket. AdaptNN uses native OpenMP search. DARTH and DABS use their serial search routines in eight worker processes for eight-core measurements, including batch dispatch, result collection, and concatenation in elapsed time. KS2 uses its serial query routine in the single-core comparison.

All methods use the same test query matrices, external batch size, CPU locations, and host. Five repetitions after a complete warmup are performed for TraCANN, HNSW, SymphonyQG, SVS/LVQ, LoRANN, RaBitQ-IVF, ScaNN, OPQ+FastScan, and KS2. For AdaptNN, DARTH, and DABS, five repetitions after a complete warmup of at most 10 seconds are performed, three repetitions after a 10-60 second warmup, and one after longer warmups. All repetitions cover the complete test set. For TraCANN, repetitions aim for a minimum duration of 0.4 seconds. If the maximum to minimum duration ratio is larger than 1.10, five repetitions with a target of 1 second are performed instead of the first set. For baseline repetitions, a minimum duration of 0.25 seconds is targeted. The number of complete test passes per repetition is determined by the warmup duration. Elapsed time is divided by the number of passes to compute QPS.

The timed portion includes query processing, native execution, binding overhead, and output allocation. Recall calculation, label access, file writing, index loading, and policy fitting take place outside the timed region. Throughput is defined as the number of queries divided by the median duration of complete queries. The tables 10 and 11 show repetition counts and QPS ranges for settings for which at least two runs were done.

E Complete Throughput Measurements

Tables 7 and 8 show achieved test recall and throughput for each nominal threshold. For each dataset, threshold, and core count, the strongest baseline is selected from among the eligible formal confirmations. The eight-core curves are in Figure 5. Table 9 summarizes direct configurations, including ranges for methods with multiple direct query settings.

Table 7: Single-core comparisons to the fastest of 11 baselines meeting each nominal threshold. Recall columns show the achieved values. Ratios are our QPS divided by baseline QPS.

Dataset	Target	Ours recall	Ours QPS	Best baseline	Recall	QPS	Ratio
SIFT1M	95%	95.575%	23,232	SymphonyQG	95.358%	20,119	1.15
SIFT1M	99%	99.125%	16,369	SymphonyQG	99.050%	10,236	1.60
SIFT1M	99.5%	99.575%	15,770	SymphonyQG	99.542%	7,324	2.15
DEEP-1M	95%	95.310%	16,192	SymphonyQG	95.010%	13,805	1.17
DEEP-1M	99%	99.000%	12,056	SymphonyQG	99.250%	6,997	1.72
DEEP-1M	99.5%	99.520%	11,046	SymphonyQG	99.570%	5,510	2.00
GIST1M	95%	95.380%	7,754	SymphonyQG	95.550%	5,383	1.44
GIST1M	99%	99.010%	4,966	SymphonyQG	99.170%	2,155	2.30
GIST1M	99.5%	99.580%	4,355	SymphonyQG	99.520%	1,659	2.62
GloVe100	95%	95.250%	4,896	SymphonyQG	95.400%	2,106	2.32
GloVe100	99%	99.008%	1,846	ScaNN	99.017%	679	2.72
GloVe100	99.5%	99.517%	1,320	ScaNN	99.500%	568	2.32

Table 8: Eight-core comparisons with the best eligible baseline. DARTH and DABS use process pools. KS2 is included in single-core results.

Dataset	Target	Ours recall	Ours QPS	Best baseline	Recall	QPS	Ratio
SIFT1M	95%	95.575%	180,110	SymphonyQG	95.358%	150,957	1.19
SIFT1M	99%	99.125%	123,036	SymphonyQG	99.050%	76,593	1.61
SIFT1M	99.5%	99.575%	117,954	SymphonyQG	99.542%	52,142	2.26
DEEP-1M	95%	95.310%	126,272	SymphonyQG	95.010%	102,372	1.23
DEEP-1M	99%	99.000%	89,336	SymphonyQG	99.070%	54,194	1.65
DEEP-1M	99.5%	99.520%	81,458	SymphonyQG	99.570%	39,761	2.05
GIST1M	95%	95.380%	53,213	SymphonyQG	95.550%	36,461	1.46
GIST1M	99%	99.010%	33,452	SymphonyQG	99.170%	14,880	2.25
GIST1M	99.5%	99.580%	29,714	SymphonyQG	99.520%	6,251	4.75
GloVe100	95%	95.250%	33,865	SymphonyQG	95.400%	16,201	2.09
GloVe100	99%	99.008%	12,052	ScaNN	99.017%	5,004	2.41
GloVe100	99.5%	99.517%	8,439	ScaNN	99.500%	4,179	2.02

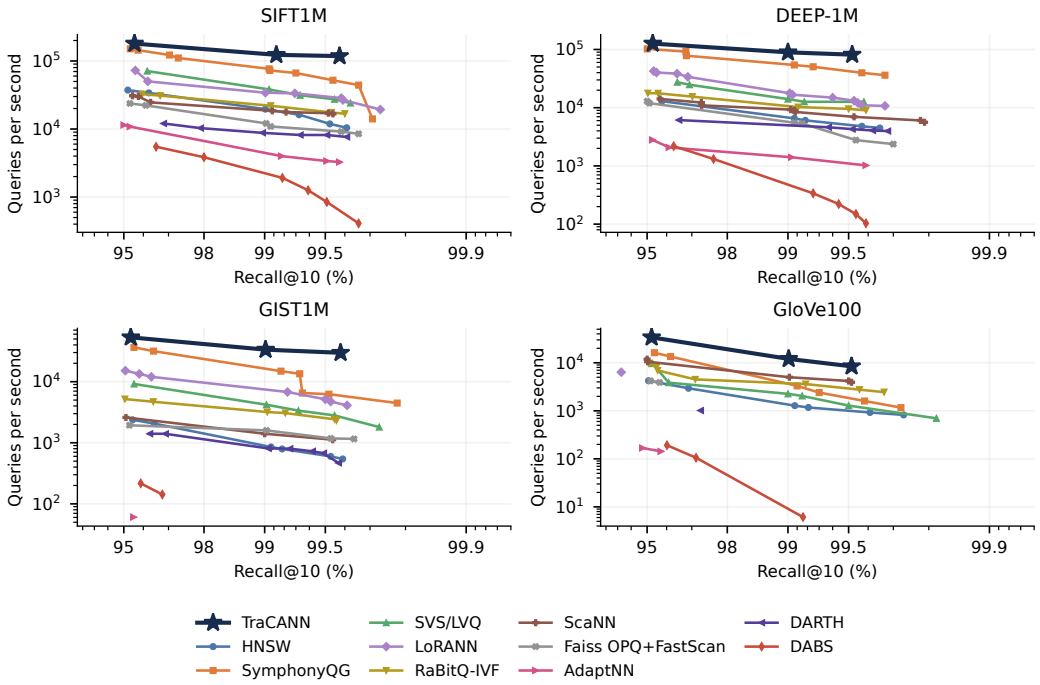


Figure 5: Eight-core throughput vs. achieved Recall@10. Curves are nondominated formal confirmations. DARTH and DABS use 8 worker processes. Other methods use native threads; external query batch size is 128.

Table 9: Measurements from direct configurations. Each interval shows the minimum and maximum across direct configurations. Single values indicate one configuration. Recall ranges use single core runs. Recall and throughput endpoints may refer to different configurations.

Dataset	Method	Recall (%)	One-core QPS	Eight-core QPS
SIFT1M	HNSW	94.15	8,261	43,310
SIFT1M	SymphonyQG	97.03	16,156	9,200
SIFT1M	SVS/LVQ	93.68	13,204	103,760
SIFT1M	LoRANN	97.80	5,341	7,013
SIFT1M	RaBitQ-IVF	86.28–99.22	360–6,851	2,709–44,655
SIFT1M	ScaNN	99.82	1,807	12,131
SIFT1M	OPQ+FastScan	23.42	10,971	78,628
SIFT1M	HNSW+KS2	74.52–99.35	5,406–24,443	N/A
SIFT1M	AdaptNN	68.17–98.98	659–8,966	2,814–39,408
SIFT1M	DARTH	97.38–99.78	1,308–2,594	6,487–11,863
SIFT1M	DABS	99.72	29.8	130
DEEP-1M	HNSW	88.68	4,746	26,500
DEEP-1M	SymphonyQG	94.56	14,178	104,167
DEEP-1M	SVS/LVQ	90.23	9,492	54,846
DEEP-1M	LoRANN	95.90	3,714	26,118
DEEP-1M	RaBitQ-IVF	83.86–99.21	218–5,112	1,862–36,438
DEEP-1M	ScaNN	99.17	1,521	4,810
DEEP-1M	OPQ+FastScan	24.41	5,368	38,373
DEEP-1M	HNSW+KS2	8.76–25.13	6,357–18,220	N/A
DEEP-1M	AdaptNN	76.97–99.36	201–3,110	961–4,461
DEEP-1M	DARTH	97.02–99.80	639–1,332	3,206–5,838
DEEP-1M	DABS	99.72	7.4	31.4
GIST1M	HNSW	71.23	1,758	8,749
GIST1M	SymphonyQG	87.59	8,159	56,526
GIST1M	SVS/LVQ	76.33	4,291	7,811
GIST1M	LoRANN	90.19	2,153	13,157
GIST1M	RaBitQ-IVF	71.31–99.31	73.8–1,784	421–8,198
GIST1M	ScaNN	71.22	566	2,544
GIST1M	OPQ+FastScan	14.66	1,276	7,432
GIST1M	HNSW+KS2	50.19–92.41	1,841–6,696	N/A
GIST1M	AdaptNN	81.09–97.96	6.2–138	29.0–344
GIST1M	DARTH	96.30–99.45	136–322	660–1,368
GIST1M	DABS	98.43	1.9	8.9
GloVe100	HNSW	73.71	6,311	35,643
GloVe100	SymphonyQG	75.87	12,018	109,021
GloVe100	SVS/LVQ	74.14	13,547	71,038
GloVe100	LoRANN	81.32	4,514	32,437
GloVe100	RaBitQ-IVF	74.08–98.92	270–4,607	1,963–31,622
GloVe100	ScaNN	93.86	2,151	14,101
GloVe100	OPQ+FastScan	19.88	13,981	98,680
GloVe100	HNSW+KS2	54.18–87.20	3,548–25,640	N/A
GloVe100	AdaptNN	67.87	500	1,288
GloVe100	DARTH	92.60–94.24	494–774	2,776–3,532
GloVe100	DABS	99.16	1.2	6.1

Table 10: Single-core throughput at the 99% threshold. Each row uses the chosen eligible configuration. Reps. is the number of timed repetitions. QPS ranges are the summaries of configurations that were measured at least twice.

Dataset	Method	Recall (%)	QPS	Reps.	Min. QPS	Max. QPS
SIFT1M	TraCANN	99.125	16,369	5	16,354	16,820
SIFT1M	HNSW	99.008	3,685	5	3,268	3,800
SIFT1M	SymphonyQG	99.050	10,236	5	8,952	10,239
SIFT1M	SVS/LVQ	99.050	6,661	5	5,810	6,720
SIFT1M	LoRANN	99.008	5,345	5	5,293	5,369
SIFT1M	RaBitQ-IVF	99.067	2,958	5	2,936	2,993
SIFT1M	ScaNN	99.067	3,542	5	3,527	3,547
SIFT1M	OPQ+FastScan	99.017	1,572	5	1,558	1,591
SIFT1M	HNSW+KS2	99.142	6,647	5	6,252	6,654
SIFT1M	AdaptNN	99.117	674	5	670	675
SIFT1M	DARTH	99.333	1,796	5	1,787	1,798
SIFT1M	DABS	99.183	401	5	400	403
DEEP-1M	TraCANN	99.000	12,056	5	11,910	12,256
DEEP-1M	HNSW	99.070	1,263	5	1,167	1,286
DEEP-1M	SymphonyQG	99.250	6,997	5	6,801	7,017
DEEP-1M	SVS/LVQ	99.000	2,528	5	2,279	2,539
DEEP-1M	LoRANN	99.000	2,356	5	2,307	2,375
DEEP-1M	RaBitQ-IVF	99.110	1,490	5	1,483	1,493
DEEP-1M	ScaNN	99.020	1,789	5	1,714	1,791
DEEP-1M	OPQ+FastScan	99.070	576	5	568	577
DEEP-1M	HNSW+KS2	99.070	166	5	165	168
DEEP-1M	AdaptNN	99.200	223	5	220	223
DEEP-1M	DARTH	99.290	883	5	878	885
DEEP-1M	DABS	99.250	63.2	3	60.5	79.3
GIST1M	TraCANN	99.010	4,966	5	4,933	5,005
GIST1M	HNSW	99.070	153	5	152	156
GIST1M	SymphonyQG	99.170	2,155	5	2,128	2,173
GIST1M	SVS/LVQ	99.020	542	5	530	545
GIST1M	LoRANN	99.000	1,075	5	1,040	1,099
GIST1M	RaBitQ-IVF	99.030	439	5	395	441
GIST1M	ScaNN	99.000	217	5	212	220
GIST1M	OPQ+FastScan	99.020	242	5	237	247
GIST1M	HNSW+KS2	99.130	601	5	592	617
GIST1M	AdaptNN	Target not reached				
GIST1M	DARTH	99.040	174	5	172	174
GIST1M	DABS	Target not reached				
GloVe100	TraCANN	99.008	1,846	5	1,838	1,859
GloVe100	HNSW	99.075	239	5	188	241
GloVe100	SymphonyQG	99.100	419	5	418	428
GloVe100	SVS/LVQ	99.000	303	5	264	303
GloVe100	LoRANN	Target not reached				
GloVe100	RaBitQ-IVF	99.183	518	5	515	519
GloVe100	ScaNN	99.017	679	5	660	681
GloVe100	OPQ+FastScan	Target not reached				
GloVe100	HNSW+KS2	99.075	392	5	386	396
GloVe100	AdaptNN	Target not reached				
GloVe100	DARTH	Target not reached				
GloVe100	DABS	99.158	1.2	1	N/A	N/A

Table 11: Eight-core throughput at 99% threshold, including repetition counts. DARTH and DABS run with eight worker processes. QPS ranges represent settings measured at least twice.

Dataset	Method	Recall (%)	QPS	Reps.	Min. QPS	Max. QPS
SIFT1M	TraCANN	99.125	123,036	5	122,065	123,860
SIFT1M	HNSW	99.008	19,591	5	19,471	19,693
SIFT1M	SymphonyQG	99.050	76,593	5	54,364	76,681
SIFT1M	SVS/LVQ	99.050	38,179	5	20,807	38,250
SIFT1M	LoRANN	99.008	34,157	5	34,137	34,204
SIFT1M	RaBitQ-IVF	99.067	21,985	5	21,599	22,268
SIFT1M	ScaNN	99.083	18,511	5	18,198	18,542
SIFT1M	OPQ+FastScan	99.017	12,079	5	10,424	12,279
SIFT1M	AdaptNN	99.175	3,997	5	3,990	4,021
SIFT1M	DARTH	99.333	8,175	5	8,146	8,569
SIFT1M	DABS	99.183	1,914	5	1,891	1,933
DEEP-1M	TraCANN	99.000	89,336	5	87,703	90,284
DEEP-1M	HNSW	99.070	6,552	5	6,494	6,564
DEEP-1M	SymphonyQG	99.070	54,194	5	36,598	54,408
DEEP-1M	SVS/LVQ	99.000	14,056	5	8,912	14,072
DEEP-1M	LoRANN	99.020	17,717	5	17,356	17,961
DEEP-1M	RaBitQ-IVF	99.110	10,251	5	10,087	10,347
DEEP-1M	ScaNN	99.030	9,271	5	8,334	9,277
DEEP-1M	OPQ+FastScan	99.070	5,519	5	5,498	5,549
DEEP-1M	AdaptNN	99.040	1,407	5	1,136	1,467
DEEP-1M	DARTH	99.370	4,615	5	4,530	4,650
DEEP-1M	DABS	99.250	338	5	328	339
GIST1M	TraCANN	99.010	33,452	5	33,154	33,567
GIST1M	HNSW	99.070	853	5	841	881
GIST1M	SymphonyQG	99.170	14,880	5	14,762	14,978
GIST1M	SVS/LVQ	99.020	4,198	5	4,076	4,236
GIST1M	LoRANN	99.230	6,776	5	6,269	6,808
GIST1M	RaBitQ-IVF	99.030	3,187	5	3,175	3,204
GIST1M	ScaNN	99.000	1,407	5	1,404	1,411
GIST1M	OPQ+FastScan	99.020	1,595	5	1,321	1,595
GIST1M	AdaptNN	Target not reached				
GIST1M	DARTH	99.040	808	5	805	816
GIST1M	DABS	Target not reached				
GloVe100	TraCANN	99.008	12,052	5	11,929	12,454
GloVe100	HNSW	99.075	1,282	5	1,266	1,293
GloVe100	SymphonyQG	99.100	3,307	5	3,234	3,320
GloVe100	SVS/LVQ	99.000	2,256	5	2,158	2,286
GloVe100	LoRANN	Target not reached				
GloVe100	RaBitQ-IVF	99.183	3,608	5	3,372	3,636
GloVe100	ScaNN	99.017	5,004	5	4,902	5,017
GloVe100	OPQ+FastScan	Target not reached				
GloVe100	AdaptNN	Target not reached				
GloVe100	DARTH	Target not reached				
GloVe100	DABS	99.158	6.1	1	N/A	N/A

F Storage and Memory

In Table 12, we report storage and process memory of the 99% policies. Index size is the serialized graph size. RSS is the memory of the search process and runtime. Peak RSS includes the index loading. Radius storage is the size of one FP32 local scale for each database vector. Tree storage is the native node payload, with 20 bytes per node in a full binary tree.

Table 12: Index storage and process memory for TraCANN. GiB and MiB are based on powers of two.

Dataset	Index (GiB)	RSS (GiB)	Peak (GiB)	Radius (MiB)	Tree (KiB)
SIFT1M	1.311	1.045	2.359	3.81	12.05
DEEP-1M	3.576	3.609	3.611	3.81	11.35
GIST1M	11.444	11.477	11.479	3.81	13.38
GloVe100	2.416	2.449	2.452	4.51	16.00

The serialized graph in SIFT contains the original FP32 rows. Loading converts the rows into bytes, and frees the FP32 graph representation, resulting in post-search RSS being lower than the serialized index size. Both representations co-exist during loading, and contribute to the maximum RSS. Other datasets keep FP32 graphs stored. Tree payload excludes spare allocator capacity and serialization overhead.

F.1 Adaptive Baseline Construction and Fitting

Table 13 shows graph construction, trajectory collection and model fitting for the adaptive termination baselines. AdaptNN fits one model for each observation threshold and fitting time is the sum of the five models. DARTH fits one recall predictor per dataset. Both methods use 4 cores for model fitting. DABS uses its distance criterion directly.

Table 13: Construction and fitting times for adaptive termination baselines. Build cores refers to the number of construction CPU cores.

Dataset	Method	Build (s)	Build cores	Collect (s)	Fit (s)
SIFT1M	AdaptNN	48.62	4	0.11	1.54
SIFT1M	DARTH	723.34	4	81.03	8.25
SIFT1M	DABS	3,947.25	1	0.00	0.00
DEEP-1M	AdaptNN	96.07	4	0.19	7.25
DEEP-1M	DARTH	1,282.21	4	87.88	10.66
DEEP-1M	DABS	4,812.63	1	0.00	0.00
GIST1M	AdaptNN	215.17	4	11.53	24.71
GIST1M	DARTH	2,528.64	4	32.24	14.88
GIST1M	DABS	9,838.11	1	0.00	0.00
GloVe100	AdaptNN	73.48	4	1.61	3.17
GloVe100	DARTH	965.49	4	66.01	10.54
GloVe100	DABS	5,642.82	1	0.00	0.00

Table 14: Storage and peak construction RSS for adaptive termination baselines. Vectors gives database storage outside the index file. Model storage sums the fitted models.

Dataset	Method	Index (GiB)	Vectors (GiB)	Model (MiB)	Peak (GiB)
SIFT1M	AdaptNN	0.611	0.000	1.358	1.388
SIFT1M	DARTH	0.730	0.000	0.297	1.249
SIFT1M	DABS	0.178	0.477	0.000	1.820
DEEP-1M	AdaptNN	1.088	0.000	1.469	2.629
DEEP-1M	DARTH	1.207	0.000	0.302	2.203
DEEP-1M	DABS	0.178	0.954	0.000	2.168
GIST1M	AdaptNN	3.711	0.000	1.682	9.447
GIST1M	DARTH	3.830	0.000	0.305	7.448
GIST1M	DABS	0.270	3.576	0.000	5.334
GloVe100	AdaptNN	0.600	0.000	1.392	1.117
GloVe100	DARTH	0.600	0.000	0.308	1.175
GloVe100	DABS	0.232	0.441	0.000	2.001

Table 14 includes database vectors stored outside of DABS graph files. AdaptNN and DARTH store vectors within their indexes. Peak RSS is per construction process. Aggregate process-tree PSS is used for eight-worker queries to account for shared index pages.