ACS Final Track B Project

Jaden Tompkins

December 2025

1 Introduction

Modern applications such as similarity search in recommendation systems and vision models rely heavily on fast approximate nearest-neighbor search over large collections of high-dimensional vectors. Pure-DRAM indexes offer excellent latency and throughput but quickly become prohibitively expensive as datasets grow into the hundreds of millions of vectors. At the same time, current SSDs provide much cheaper capacity but introduce orders-of-magnitude higher latency and complex I/O behavior. This project envisions a future where SSD speeds reach current DRAM capabilities, allowing for memory and storage to blend. In such a computer system, algorithms such as ANN will need to be executed utilizing SSDs.

1.1 Motivation and Problem Setting

We focus on ANN workloads over SIFT1M and synthetic Gaussian datasets with tens to hundreds of thousands of base vectors and thousands of queries. The goal is to sustain high recall at fixed k while delivering competitive query throughput and controlling system cost. We study three families of solutions under a unified simulator: an in-memory graph index, a tiered DRAM+SSD cache hierarchy, and an ANN-in-SSD design that executes much of the search logic inside a storage-attached block-graph index. This project is not meant to implement an industry ready solution or even a true prototype. Rather, it explores the possible performance of ANN utilizing various levels of SSD incorporation, comparing to current solutions, and investigating what performance characteristics need to look like for each to be viable.

1.2 Contributions

Our contributions are as follows:

- We implement a DRAM-only baseline and a tiered DRAM+SSD design that share a common ANN index structure but differ in where data resides and how cache hits and misses are handled. These are mainly to show how our implementations may be unoptimized compared to industry standard and so certain performance needs to be taken with a grain of salt, but also provide additional data to compare to.
- We design an ANN-in-SSD block-graph index with tunable parameters (vectors per block, portal degree, and max_steps) and instantiate several hardware levels that approximate increasingly capable SSDs.
- We conduct a systematic evaluation across Experiments 0–12, measuring recall, throughput, I/O amplification, cost, and scalability. The experiments progress from baselines, to cache behavior, cost-performance trade-offs, and unified comparisons.
- We present recall-matched comparisons that highlight where ANN-in-SSD and tiered designs can approach or exceed DRAM baselines at substantially lower cost (with a bit of hand-wavy performance characteristics alongside a concrete proof-of-concept).

2 Background and Existing Solutions

2.1 Approximate Nearest Neighbor Search

Nearest-neighbor search asks for the k closest vectors to a query under a chosen distance metric (here, Euclidean distance). Exact search quickly becomes infeasible at scale, so most practical systems use approximate structures that trade some loss in recall for large gains in throughput. Throughout this report we use recall@k, search QPS, total QPS (including build time), effective QPS (including modeled device time), and modeled device time per query as our primary metrics. To further note, as we are merely simulating SSD behavior, effective QPS takes the device performance into effect, giving an equivalent to what QPS can be expected from a device implementing our solution. This gives an equal footing to compare solutions.

2.2 HNSWlib

HNSWlib is a widely used implementation of hierarchical navigable small world graphs for ANN search. It builds a multi-layer graph and uses a greedy search strategy to quickly converge on high-quality neighbors. Tuning efConstruction and efSearch lets us trade build cost for recall and query latency. We use HNSWlib as a strong DRAM-only upper bound in Experiment 0 and as a comparison point in later experiments.

2.3 DiskANN and IO-Aware ANN

Recent systems such as DiskANN extend ANN search to SSD-backed indexes by organizing vectors into blocks and carefully controlling I/O during search. These systems typically maintain a small DRAM-resident graph that routes queries to promising blocks on SSD and stop the search once additional I/O is unlikely to improve recall. Our ANN-in-SSD design is inspired by these ideas but targets a flexible simulator in which we can vary hardware parameters and search-time knobs such as max_steps.

2.4 Tiered Storage and Caching

Many production systems use a tiered architecture in which a DRAM cache fronts a larger SSD or HDD store. Cache size and policy (e.g., LRU or LFU) directly determine how often queries can be served from memory and how much device work must be issued. Experiments 2–4 and 6 explore how cache fraction and policy choices influence recall, throughput, and I/O amplification in our simulator.

3 System Design and Implemented Solutions

3.1 Solution 1: DRAM Baseline

Our DRAM baseline builds an in-memory HNSW-style graph over the base vectors. Queries start from an entry point and greedily traverse the graph, maintaining a candidate set until the k nearest neighbors are identified. This solution provides a simple upper bound on performance when the entire index fits comfortably in memory.

3.1.1 Index Structure and Search Algorithm

Vectors are stored in a contiguous array, while adjacency lists encode edges between graph nodes. The search procedure maintains a small priority queue of promising nodes and iteratively explores neighbors until convergence. The same implementation is reused across DRAM-only and tiered experiments to isolate the effects of storage.

3.1.2 Implementation Details

We use a multi-threaded implementation with configurable graph parameters and batch size. The specific efSearch and efConstruction values used in Experiments 0 and 1 were chosen to balance recall and build/search cost, and are held fixed across competing configurations.

3.2 Solution 2: Tiered DRAM+SSD

The tiered solution keeps the HNSW graph structure in DRAM but stores vector payloads behind a StorageBackend abstraction (in our experiments, an in-memory backing store paired with a modeled SSD device time). A DRAM cache keyed by vector ID holds recently accessed vectors. During graph traversal, a cache hit returns the vector from memory, while a miss consults the backing store, records a modeled SSD read, and inserts the vector into the cache. This architecture reduces the modeled DRAM footprint while approaching DRAM-like recall, at the cost of additional device time when the working set is not cacheable.

3.2.1 Cache Layout and Policies

Each cache entry corresponds to a single vector payload, so cache capacity is expressed in number of cached vectors (and reported as a fraction of the index size). We experiment with LRU and LFU replacement policies and vary the cache capacity across experiments. Experiments 2–4 show how these choices impact hit rate, I/O per query, and effective QPS.

3.2.2 IO and Latency Model

The simulator models a multi-channel SSD with configurable base latency, internal bandwidth, and queue depth. Each cache miss triggers an asynchronous read, and device time per query is computed from the sum of issued reads and modeled service times. This allows us to explore how IO amplification and SSD characteristics affect end-to-end throughput.

3.3 Solution 3: ANN-in-SSD

ANN-in-SSD moves more of the search computation into the storage layer by coupling a block-graph index with a parameterized search procedure. Queries traverse the graph across blocks, issuing SSD reads as needed, and terminate after a configurable number of steps. This design exposes explicit knobs for trading recall against I/O work and latency.

3.3.1 Index Layout and Block Graph

Vectors are grouped into blocks of K vectors. The simulator computes a centroid for each block and constructs an inter-block graph by linking each block to its nearest centroid neighbors (controlled by portal_degree), augmented with a ring backbone for global connectivity. These inter-block links serve as coarse-grained portal edges for routing. During search, the model reads blocks as needed and scans vectors within the visited blocks to refine candidates.

3.3.2 Search Algorithm and Stopping Condition

The ANN-in-SSD search algorithm starts from an entry block and follows portal edges while maintaining a small candidate set of promising blocks. At each step it may issue additional SSD reads to refine candidates. The max_steps parameter bounds the number of block expansions, providing a direct mechanism to limit I/O while preserving much of the recall seen in full-scan configurations.

3.3.3 Hardware Levels L0–L3

To explore the impact of media characteristics, we define four hardware levels (L0–L3) with progressively lower latency, higher bandwidth, and larger queue depths. Specifically, each level provides an additional level of support for ANN-like operations in-SSD, assuming that more general hardware is more likely to be implemented (L0) than highly-optimized-for-ANN-specifically hardware (L3). Specific vision for each level is as follows, but is modeled simply by faster speeds and greater efficiency.

- L0 (DRAM-speed SSD): ANN is executed by the CPU utilizing a very high speed SSD with the built-in organizational structures. Here, the CPU does all the thinking, with some structural help from the SSD.

- L1 (ANN-in-SSD): ANN execution occurs on the SSD controller in the on-drive DRAM, allowing for high speed ANN on-demand.
- L2 (parallel ANN-in-SSD): Compute is expanded into parallel SIMD blocks allowing for many paths to be searched at once in-SSD. These blocks are placed near or within the NAND channels for very high speed access and integration.
- L3 (optimized, parallel ANN-in-SSD): Here, specific ALUs are also integrated into the compute blocks, allowing vector operations to be done en-mass, in parallel, close to the NAND. This gives high speed access and high speed computation with hardware extremely optimized specifically for ANN.

4 Experimental Methodology

4.1 Datasets

We evaluate on two representative datasets. SIFT1M consists of 128-dimensional visual descriptors with a public ground-truth k-NN index. The synthetic Gaussian dataset is generated by sampling points from a high-dimensional Gaussian distribution, which provides a controlled setting with simpler structure. For both datasets we create subsets with different numbers of base vectors and queries to stress scalability and cache behavior.

4.2 Metrics

Our primary accuracy metric is recall@k, computed by comparing the returned neighbors against precomputed ground truth. Throughput is reported both as search QPS (queries per second during the search phase) and total QPS, which amortizes index build time across all queries. For SSD-backed designs we also report effective QPS, which incorporates modeled device time into the search phase. To reason about I/O efficiency we track reads per query, bytes per query, and modeled device time per query.

4.3 Hardware and Simulator Parameters

All experiments are run in a custom simulator that models DRAM access as effectively free and parameterizes SSD devices by base read latency, internal bandwidth, number of channels, and queue depth. These parameters are chosen to roughly approximate modern NVMe SSDs but can be varied in sensitivity studies such as Experiment 5. For the cost-performance analysis we adopt simple per-GB pricing assumptions for DRAM and SSD capacity.

4.4 Workloads and Query Configuration

Unless otherwise noted, each experiment uses thousands of queries drawn from the standard evaluation sets for SIFT1M or the synthetic dataset. We fix $k = 10$ throughout and typically use between 2,000 and 10,000 queries per configuration unless a particular experiment explicitly varies these values. Queries are processed in batches, and the simulator accounts for both index build time and query processing time. Tiered and ANN-in-SSD configurations are evaluated after warming the cache where appropriate so that steady-state behavior is captured.

4.5 Reproducibility and Artifacts

All experiments are driven by per-experiment scripts under `experiments/Experiment*/scripts/`. Each run emits JSON results under the corresponding `results/raw/` directory, and each experiment has a Python analysis script that reads these JSON files and produces plots under `results/plots/`. The figures included in this report are copied into `report/plots/` so the LaTeX build is self-contained.

4.6 Limitations and Threats to Validity

Our simulator and implementations are intentionally simplified, so the absolute performance numbers should be interpreted primarily as trends. In particular, the tiered configuration uses an in-memory backing store and charges a modeled SSD service time on cache misses, which does not capture OS/page-cache behavior or true host memory pressure. The SSD model assumes idealized parallelism (channels $\times$ queue depth) and does not simulate detailed per-channel scheduling or contention, so device time is a coarse estimate. Our reported latency percentiles measure host-side search time; modeled device time is reported separately and only reflected in effective QPS. The ANN-in-SSD results additionally rely on a simplified centroid block graph and (in cheated mode) analytic compute-time estimates. In cheated mode we replace host-side search timing with analytic compute-time estimates and assume ideal overlap with device time, whereas the faithful simulator explicitly simulates host search and adds it to modeled SSD service time; the latter configuration is used for the core recall-matched comparisons in Section 5. Put simply, faithful mode simulates each operation that would be taken by our future SSD to showcase a concrete proof-of-concept, but due to this, performs incredibly slow in realtime. To compensate, cheated mode was added that assumes ANN will be computed in this way, but simply calculates its performance rather than executing it, allowing much longer runs to be simulated and analyzed. Care was taken to ensure that results from both modes are equivalent so that cheated mode can be trusted to be representative of our solution.

4.7 Experiment Summary

Table 1 summarizes the experiments performed in this work. The early experiments establish DRAM and tiered baselines, the middle experiments explore I/O amplification, SSD characteristics, and cost, and the later experiments study the ANN-in-SSD design space and unified recall-matched comparisons.

Table 1: Summary of experiments and the primary questions they answer.

Experiment	Name	Primary question / focus
0	HNSWLIB baseline	What is the DRAM-only upper bound for recall and QPS using a tuned HNSWlib configuration?
1	DRAM baseline	How does the in-house DRAM implementation perform in terms of recall, QPS, and latency percentiles?
2	Tiered vs DRAM	How much recall/QPS do we lose or gain when introducing a DRAM+SSD cache hierarchy?
3	Cache policies	How do LRU and LFU policies affect hit rate, IO, and effective QPS?
4	IO amplification	How do reads/query, bytes/query, and device time/query scale with cache fraction?
5	SSD sensitivity	How sensitive is performance to SSD latency and bandwidth profiles?
6	Cost-performance	Under a simple cost model, when are DRAM, tiered, and ANN-in-SSD cost-effective?
7	Scaling	How do build time and effective QPS scale with dataset size for DRAM and tiered solutions?
8	SOTA comparison	How do our solutions compare to HNSWlib at similar recall and dataset sizes?
9–10	ANN-in-SSD design space	(From later experiments) How do ANN-in-SSD parameters (K per block, portal degree, max_steps) affect recall/QPS?
11	ANN-in-SSD hardware levels	How do different SSD hardware levels impact ANN-in-SSD performance and device time?
12	Unified comparison	At matched recall targets, how do DRAM, tiered, HNSWlib, and ANN-in-SSD compare across datasets?

5 Results

This section presents the main quantitative results. We first establish DRAM and HNSWlib baselines, then study how tiered caching affects performance and I/O amplification. We next explore SSD sensitivity and cost-performance trade-offs, before examining the scalability of our solutions and their comparison to state-of-the-art ANN systems. Finally we turn to the ANN-in-SSD design space and unified recall-matched views across all methods. Table 1 provides a roadmap from Experiments 0–12 to the questions they answer and the figures that follow.

5.1 Experiment 0: HNSWLIB DRAM Baseline

Experiment 0 measures the performance of tuned HNSWlib configurations on our datasets. Across a range of efSearch values HNSWlib achieves near-perfect recall while delivering high query throughput, providing a useful upper bound for DRAM-based solutions. Later experiments evaluate how closely our implementations and SSD-based designs can approach this baseline. This behavior is summarized in Figures 1 and 2.

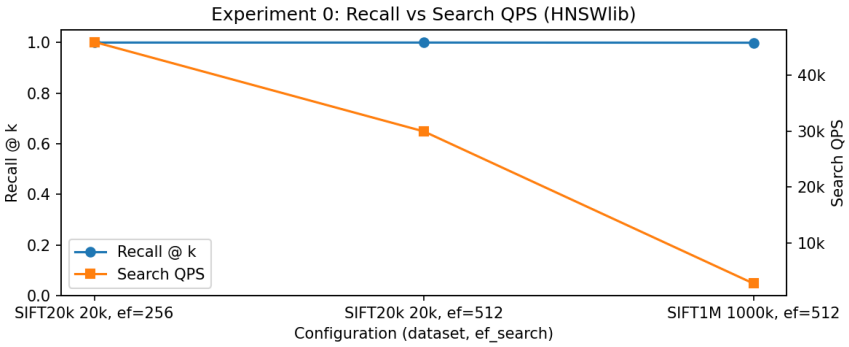

Figure 1: Experiment 0: recall vs search QPS for several HNSWlib configurations.

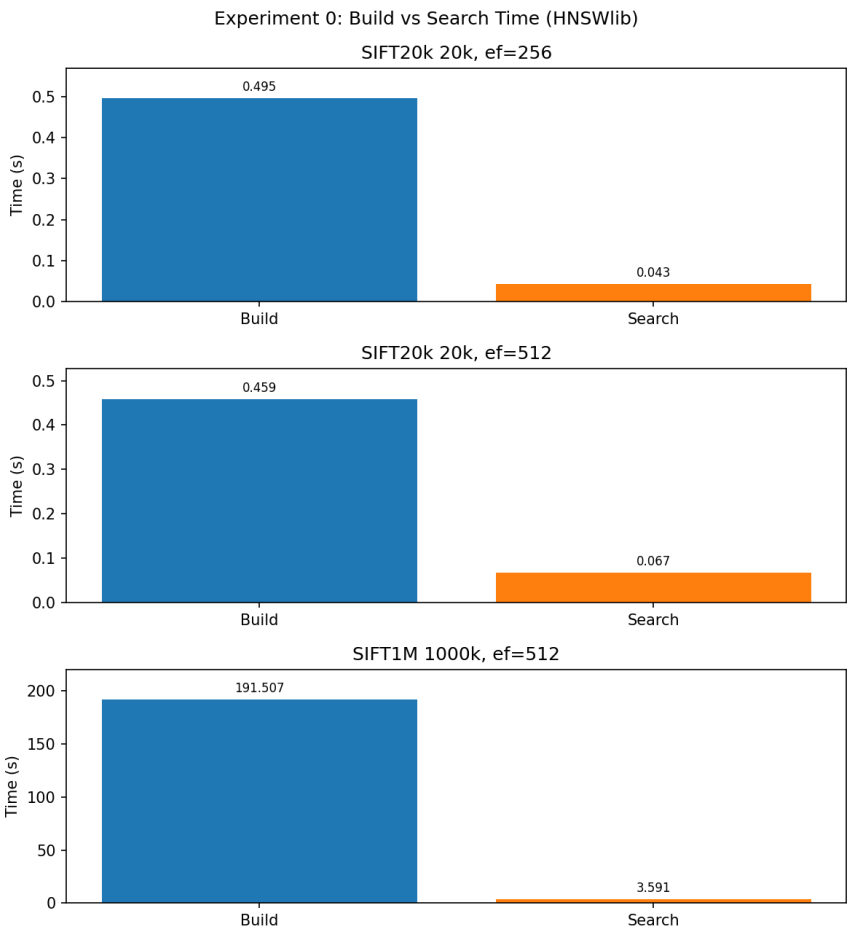

Figure 2: Experiment 0: build and search time per HNSWlib configuration.

5.2 Experiments 1–2: DRAM Baseline and Tiered vs DRAM

Experiments 1 and 2 establish the behavior of our DRAM-only implementation and the impact of introducing a DRAM+SSD cache. The DRAM baseline confirms that our in-house index can match the accuracy of HNSWlib while exposing more detailed latency metrics. Under the tiered configuration, recall remains near the DRAM baseline, but effective QPS is dominated by modeled device time: as cache capacity increases, reads/query and device time drop, improving effective throughput.

5.2.1 Experiment 1: DRAM Baseline

Figures 3 and 4 show the recall/QPS and latency percentiles for DRAM-only runs over the efSearch sweep.

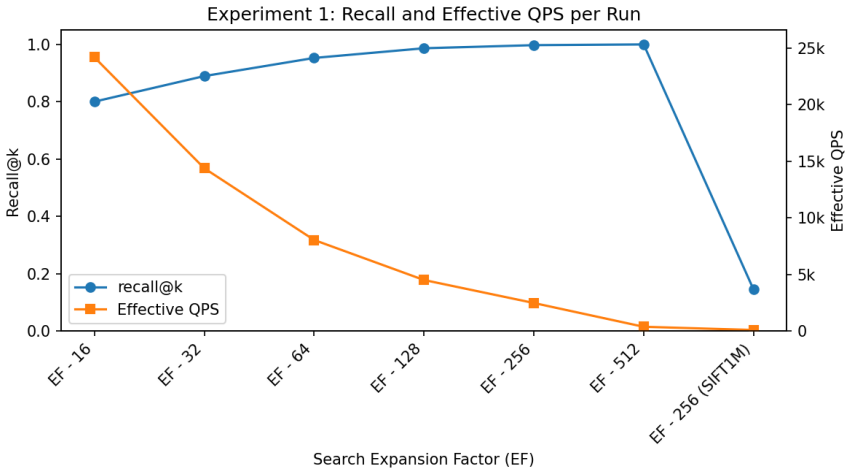

Figure 3: Experiment 1: recall and effective QPS for DRAM-only runs as we sweep the search expansion factor ef (x-axis labels of the form EF–16, EF–32, EF–64,...). All but one point correspond to the synthetic Gaussian workload; the EF–256 (SIFT1M) label marks the SIFT1M DRAM run.

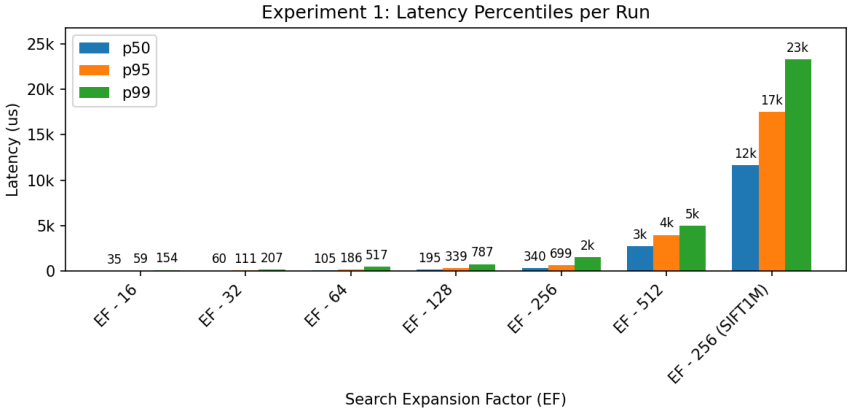

Figure 4: Experiment 1: latency percentiles (p50, p95, p99) for DRAM-only runs over the same ef sweep (x-axis labels EF–16, EF–32, EF–64,...). As in the recall/QPS plot, the EF–256 (SIFT1M) label denotes the single SIFT1M point, while the other points are synthetic Gaussian.

5.2.2 Experiment 2: Tiered DRAM+SSD vs DRAM

Figures 5 and 6 highlight how recall, effective QPS, and I/O per query change as we vary the tiered cache fraction.

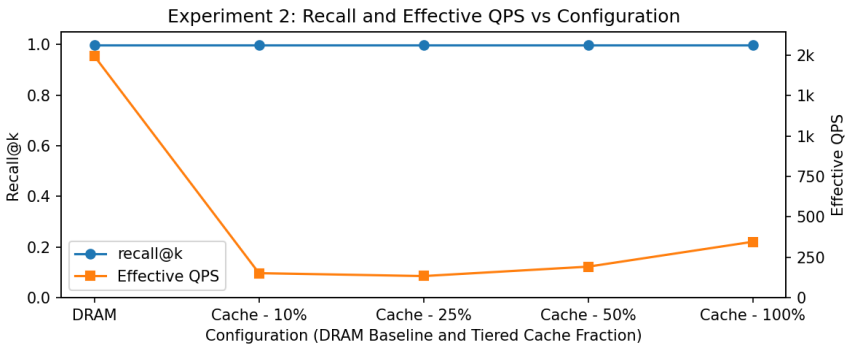

Figure 5: Experiment 2: recall and effective QPS for DRAM and tiered configurations as we vary the DRAM-only baseline and tiered cache fractions (x-axis labels "DRAM" and "Cache – 10%", "Cache – 25%", "Cache – 50%", "Cache – 100%").

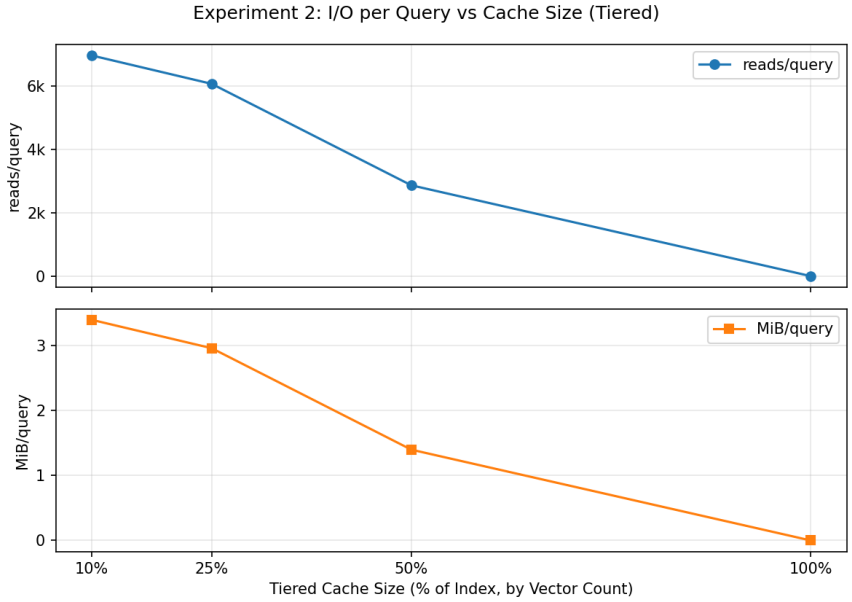

Figure 6: Experiment 2: reads/query and MiB/query vs cache percentage for tiered runs.

5.3 Experiments 3–4: Cache Policies and IO Amplification

5.3.1 Experiment 3: Cache Policies

Experiment 3 compares LRU and LFU cache replacement policies under a constant cache fraction. We find that both policies achieve similar recall, but their hit rates and effective QPS can diverge depending on the workload. These results motivate the choice of policy in the remaining experiments and are visualized in Figures 7 and 8.

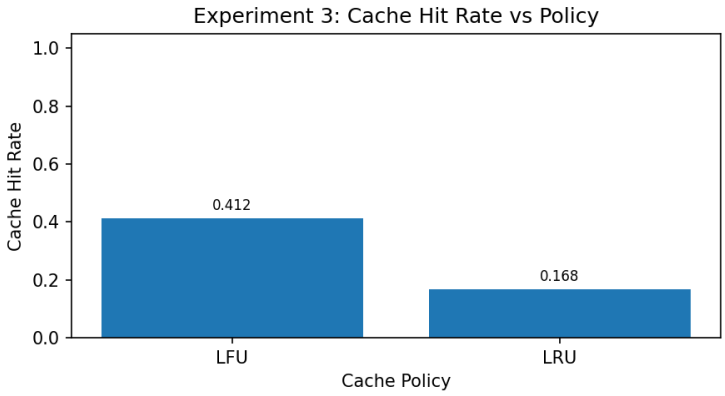

Figure 7: Experiment 3: cache hit rate vs policy (LRU vs LFU).

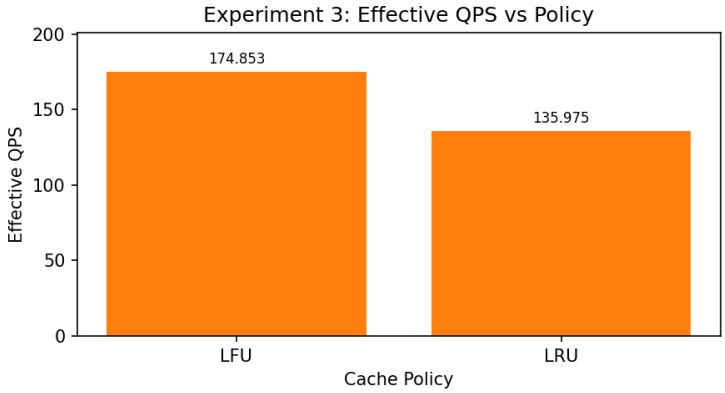

Figure 8: Experiment 3: effective QPS vs cache policy.

5.3.2 Experiment 4: IO Amplification vs Cache Fraction

Experiment 4 varies the cache fraction while measuring reads per query, bytes per query, and modeled device time. As expected, increasing the cache size reduces I/O and device time sharply until most of the hot working set fits in memory, after which returns diminish. These trends quantify the memory–I/O trade-off for the tiered design and are shown in Figures 9, 10, and 11.

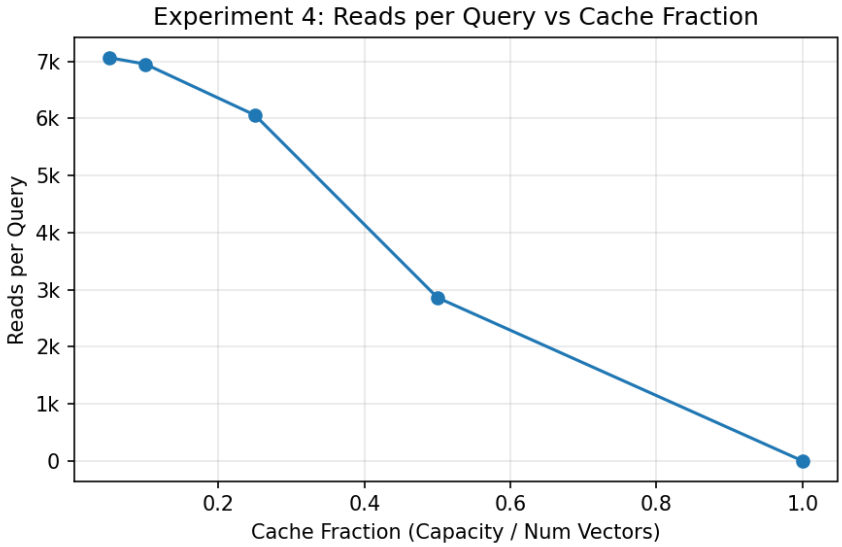

Figure 9: Experiment 4: reads per query vs cache fraction.

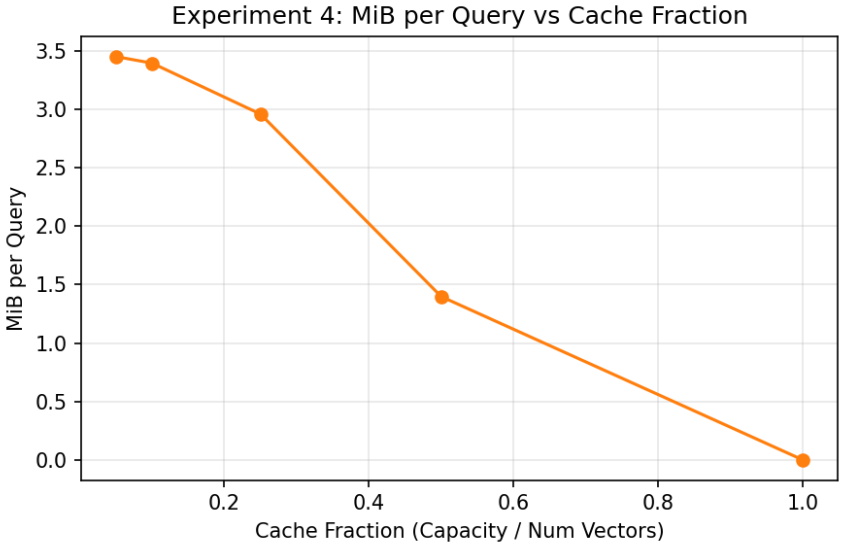

Figure 10: Experiment 4: bytes per query vs cache fraction.

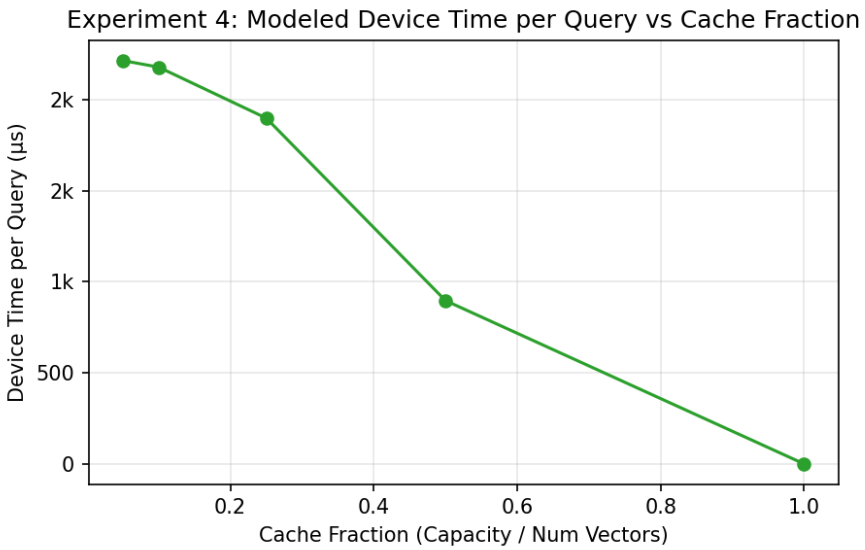

Figure 11: Experiment 4: modeled device time per query vs cache fraction.

5.4 Experiment 5: SSD Sensitivity

Experiment 5 studies how the tiered design responds to different SSD latency and bandwidth profiles. Moving to faster media improves both effective QPS and device time per query, but the benefit depends on how often queries miss in the cache. The results highlight which SSD parameters matter most for this workload and are summarized in Figures 12 and 13.

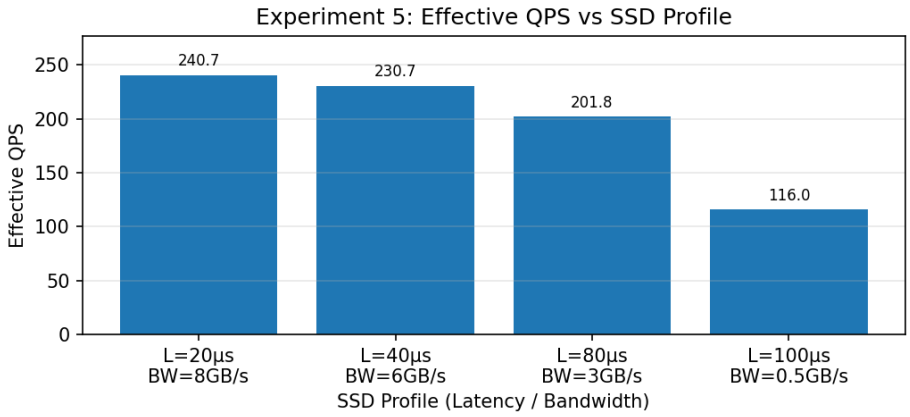

Figure 12: Experiment 5: effective QPS vs SSD profile (latency/bandwidth).

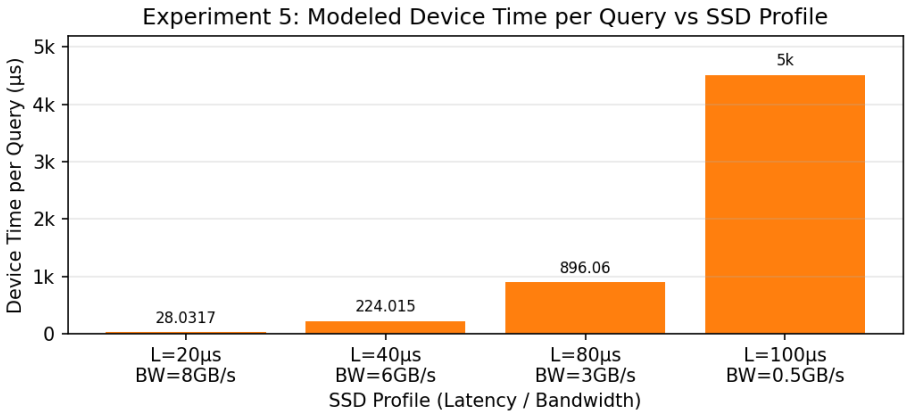

Figure 13: Experiment 5: modeled device time per query vs SSD profile.

5.5 Experiment 6: Cost–Performance Trade-off

Experiment 6 combines performance results with a simple DRAM/SSD cost model to compare the solutions on cost per unit of effective throughput. For each configuration we approximate the index size as $N_{\text{base}} \times d \times 4$ bytes (float32 vectors), assume DRAM costs \$10/GB and SSD capacity for tiered runs costs \$1/GB, and scale ANN-in-SSD SSD prices by hardware level (L0–L3: \$0.8, \$1.0, \$1.5, \$2.0 per GB). DRAM runs place the full index in memory, tiered runs keep only a cache-fraction of the index in DRAM while storing the full index on SSD, and ANN-in-SSD runs assume a small host DRAM footprint ($\approx 10\%$ of the index) with the full index on the ANN-SSD device. Total cost is then $C = C_{\text{DRAM}} + C_{\text{SSD}}$ and we report cost-per-QPS as C/effective_QPS. Under this model DRAM provides excellent QPS but at high cost, while tiered and ANN-in-SSD configurations offer more favorable cost-per-QPS in many regimes, as shown in Figure 14. Figure 15 uses the same model but sweeps the assumed ANN-in-SSD media price to show when Solution 3 becomes competitive with the best DRAM/tiered configuration. These results help identify operating points where SSD-heavy designs are economically attractive and how aggressively ANN-in-SSD must be priced to win.

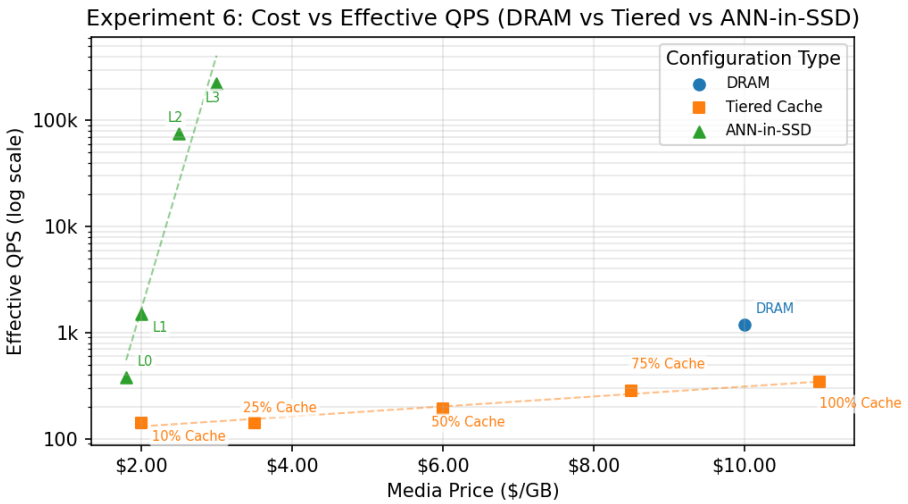

Figure 14: Experiment 6: total cost vs effective QPS for DRAM, tiered, and ANN-in-SSD configurations.

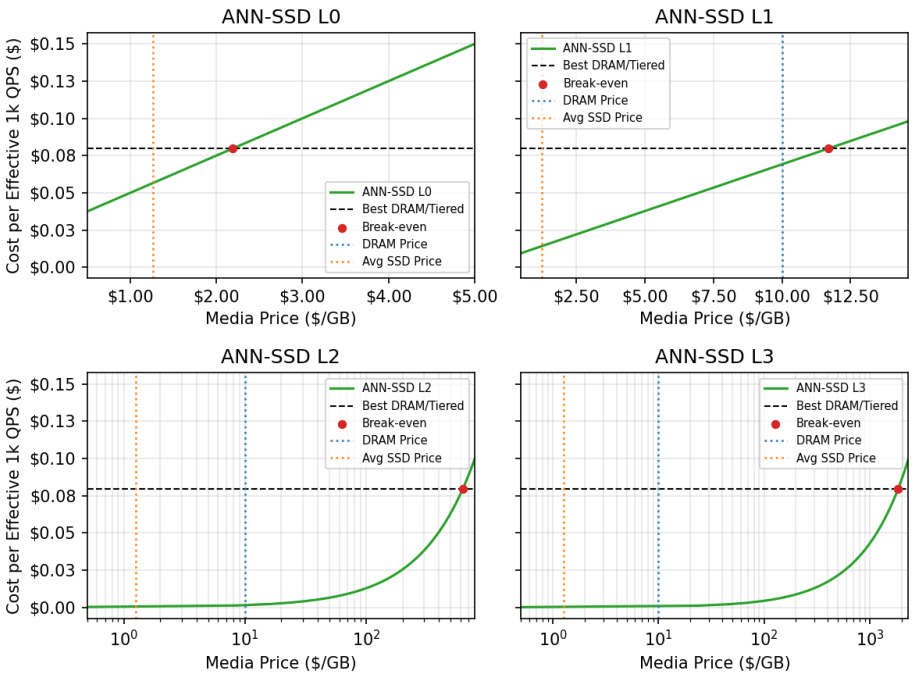

Figure 15: Experiment 6: ANN-in-SSD cost-per-QPS sweep vs assumed media price, compared to DRAM/tiered baselines. Note: L0 & L1 are scaled linearly while L2 & L3 have a logarithmic x-axis.

5.6 Experiment 7: Scaling with Dataset Size

Experiment 7 examines how build time and effective QPS scale as we increase the number of base vectors. Both DRAM and tiered solutions exhibit roughly linear growth in build time and gradual declines in effective QPS as datasets grow. The curves in Figures 16 and 17 illustrate the practical limits of each design when targeting much larger corpora.

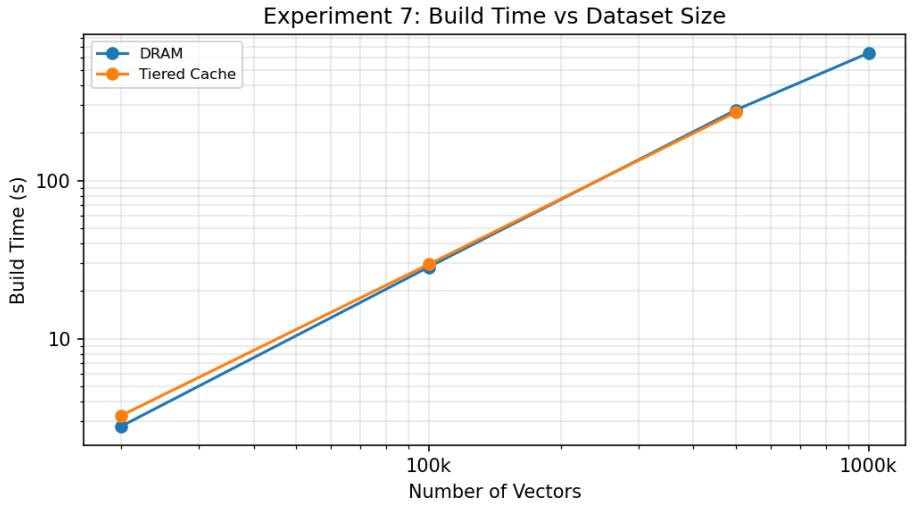

Figure 16: Experiment 7: build time vs number of base vectors (log–log).

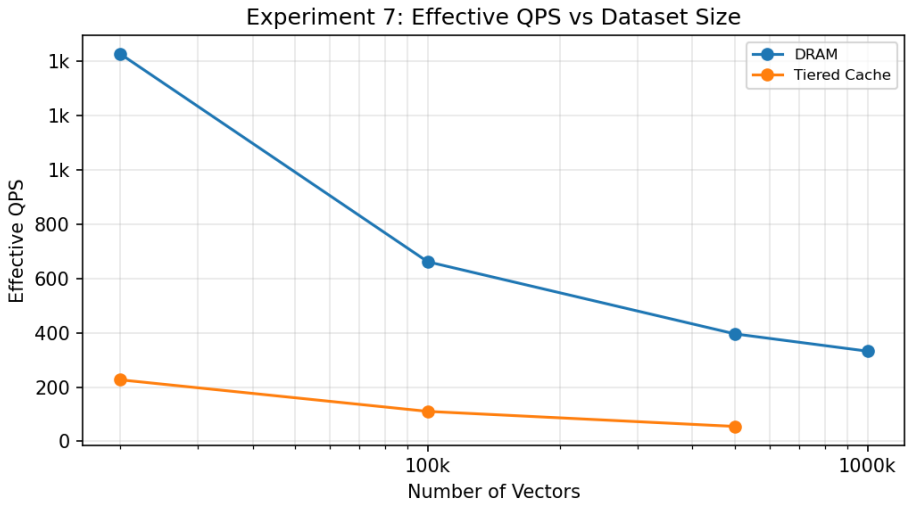

Figure 17: Experiment 7: effective QPS vs number of base vectors.

5.7 Experiment 8: Comparison with State-of-the-Art (HNSWlib)

Experiment 8 directly compares our DRAM implementation to HNSWlib on a common SIFT1M configuration at several dataset sizes and a fixed recall target. HNSWlib typically achieves slightly higher throughput at comparable recall, reflecting its highly optimized in-memory design, but our DRAM baseline remains within a reasonable factor. This experiment validates that the simulator and implementations capture realistic trade-offs, as summarized in Figures 18 and 19.

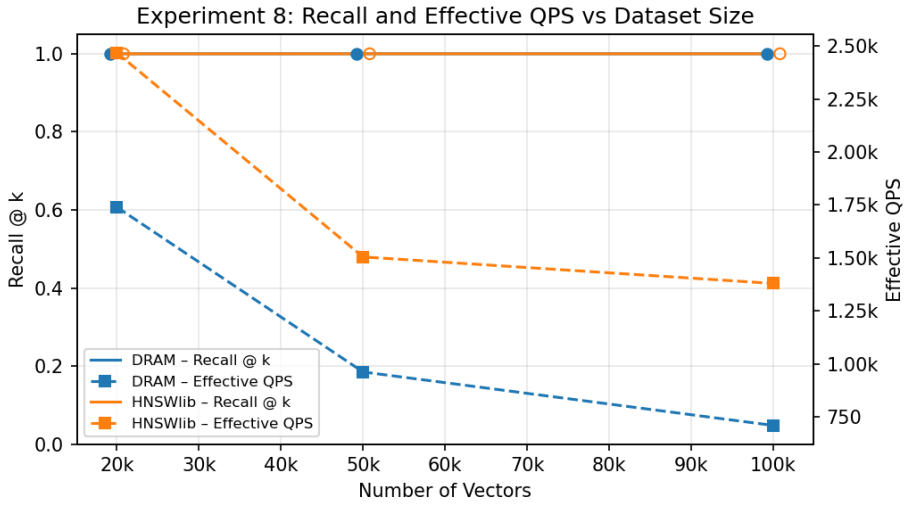

Figure 18: Experiment 8: recall @ k and effective QPS vs number of base vectors for HNSWlib and our DRAM implementation (dual y axes).

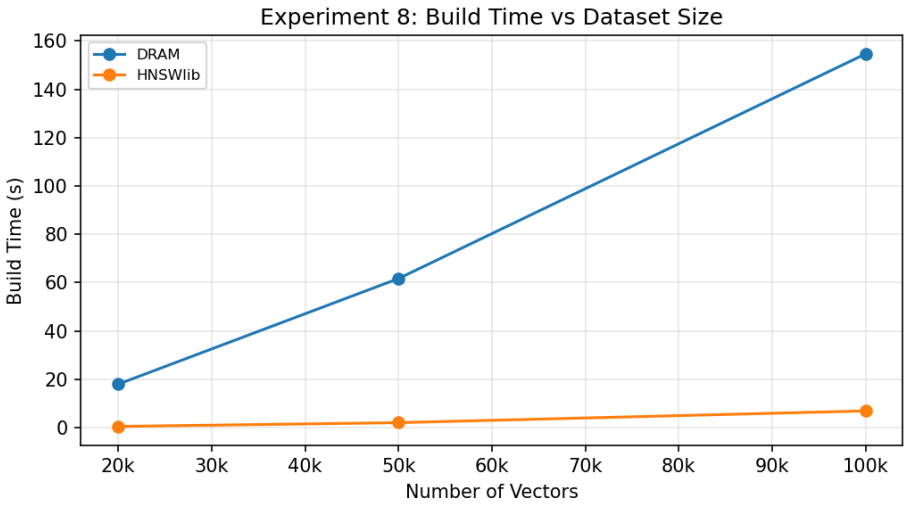

Figure 19: Experiment 8: build time vs number of base vectors for HNSWlib and our DRAM implementation.

5.8 Later Experiments: ANN-in-SSD Design Space and Unified Comparison

The remaining experiments focus on ANN-in-SSD and the final unified comparison across all methods. Experiment 9 contrasts ANN-in-SSD with DRAM and tiered baselines, Experiment 10 explores the ANN-in-SSD design space, Experiment 11 studies hardware levels, and Experiment 12 presents recall-matched views across all solutions. Together they show when ANN-in-SSD can deliver competitive or superior cost-normalized performance. Quantitatively, Experiment 9 on the synthetic dataset shows ANN-SSD L3 reaching about 28,000 effective QPS at a 0.85 recall target compared to roughly 3,800 for DRAM and 700 for the tiered configuration, while the unified recall-matched view in Experiment 12 highlights that at 0.99 recall on SIFT1M with 100k base vectors, ANN-SSD L3 achieves around 5,700 effective QPS versus approximately 740 and 1,100 for DRAM and HNSWlib. Note that this is assuming highly optimized performance for L3.

5.8.1 Experiment 9: ANN-in-SSD vs Tiered vs DRAM

Experiment 9 compares ANN-in-SSD against DRAM and tiered solutions on the synthetic dataset. Figure 20 illustrates how ANN-in-SSD traces out a different region of the recall/QPS space, offering additional design points that are difficult to realize with purely DRAM-based indexes, and Table 3 summarizes recall-matched effective QPS for several targets.

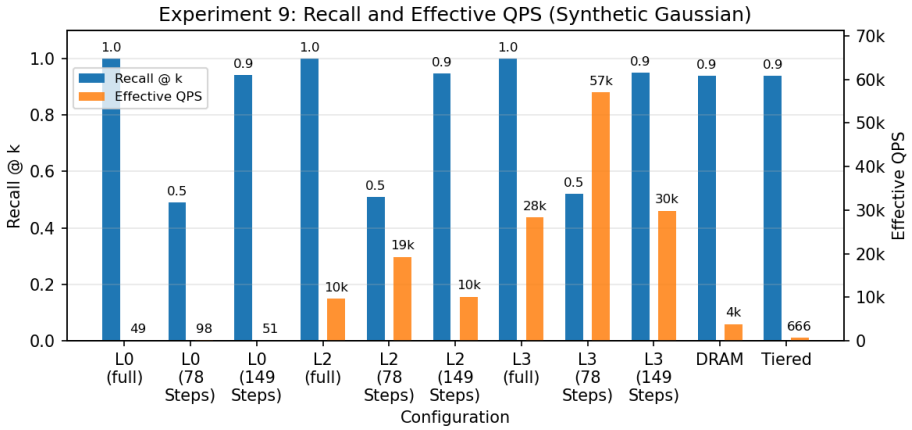

Figure 20: Experiment 9: recall @ k and effective QPS for DRAM, tiered, and ANN-in-SSD configurations on the synthetic Gaussian dataset (bar chart).

5.8.2 Experiment 10: ANN-in-SSD Design Space

Experiment 10 sweeps the ANN-in-SSD design parameters, most notably the max_steps limit and portal degree P. The results show how increasing max_steps steadily improves recall at the cost of more I/O and lower effective QPS, while higher portal degree changes the shape of this trade-off. The detailed K=128 sweep highlights a rich set of operating points around recall targets such as 0.9 and 0.95. These trends are shown in Figures 21–27.

Figure 21: Experiment 10: effective QPS and recall vs max_steps for portal degree $P = 1$.

Figure 22: Experiment 10: effective QPS and recall vs max_steps for portal degree $P = 2$.

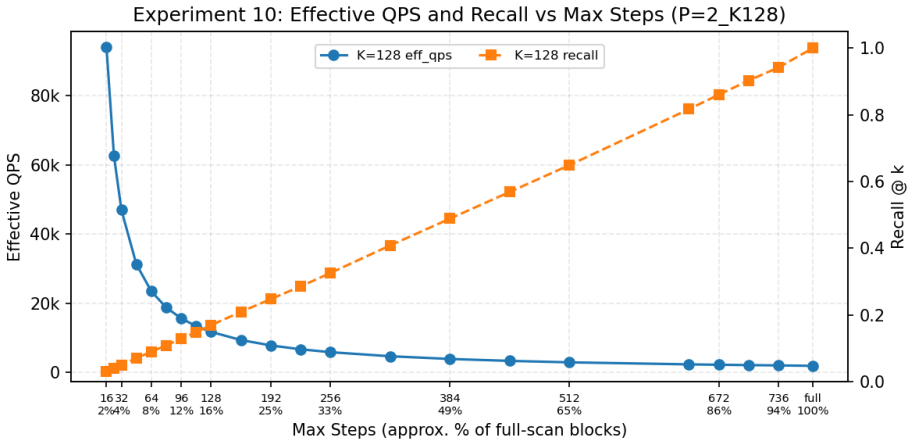

Figure 23: Experiment 10: detailed max_steps sweep for $K = 128$, $P = 2$ showing the DRAM–ANN-in-SSD trade-off.

Figure 24: Experiment 10: effective QPS and recall vs max_steps for portal degree $P = 4$.

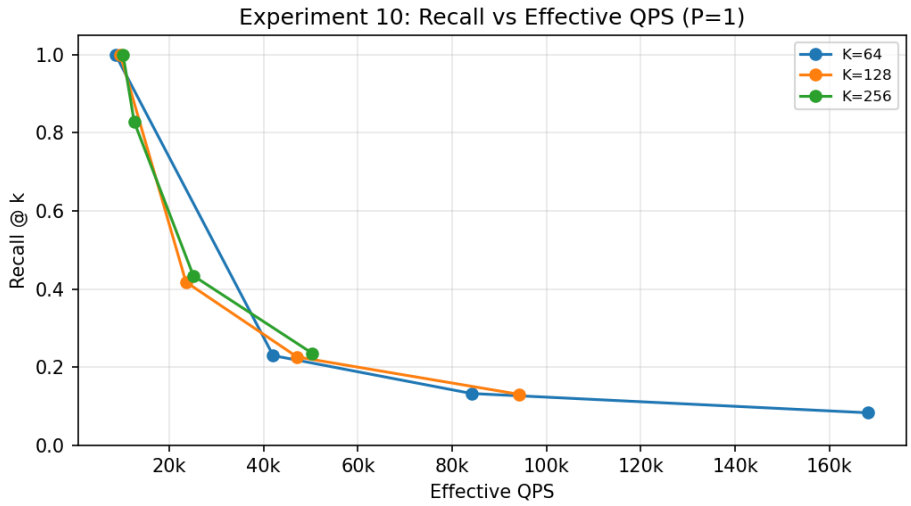

Figure 25: Experiment 10: recall vs effective QPS for $P = 1$ across ANN-in-SSD design points.

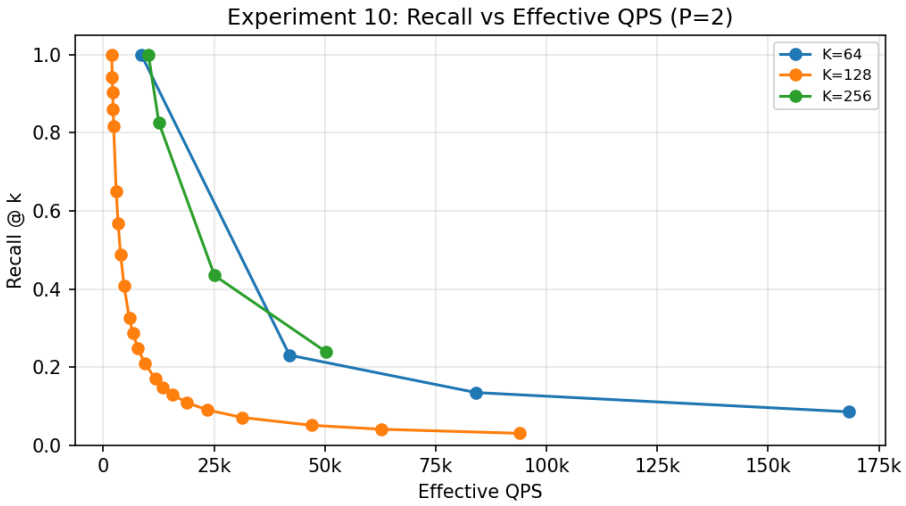

Figure 26: Experiment 10: recall vs effective QPS for $P = 2$ across ANN-in-SSD design points.

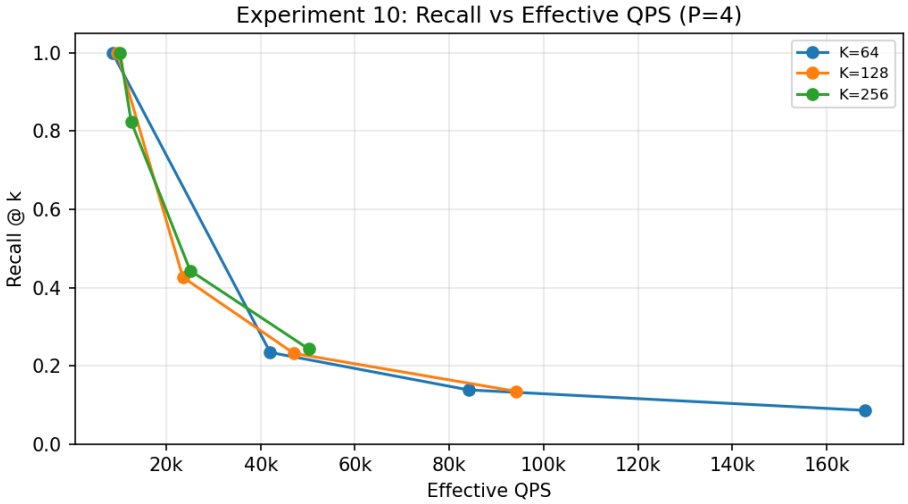

Figure 27: Experiment 10: recall vs effective QPS for $P = 4$ across ANN-in-SSD design points.

5.8.3 Experiment 11: ANN-in-SSD Hardware Levels

Experiment 11 varies the assumed SSD hardware level while keeping the *fraction of blocks visited per query* approximately fixed. We choose a reference configuration at $N = 20$k base vectors and 20 search steps and scale `max_steps` proportionally with the number of blocks as N grows. This keeps recall clustered around 0.13–0.18 across dataset sizes while letting effective QPS reflect how the same amount of work per query runs on different hardware levels. Note that this recall is very poor, but behavior scales proportionately and recall isn't significant for the results of this experiment.

Figure 28 shows the canonical effective QPS as we scale N_{base} from 5k to 80k for all four levels under approximately constant recall. Because we increase `max_steps` in proportion to the number of blocks, throughput drops with larger datasets rather than remaining perfectly flat.

To make per-level trends easier to see, we also split the same data into a 2x2 grid of subplots in Figure 29, with one panel per hardware level.

To more directly compare hardware sensitivity, we then fix $N_{\text{base}} = 20\text{k}$ and plot effective QPS and modeled device time per query (in milliseconds) as a function of hardware level in Figure 30. This dual-axis bar chart highlights the benefits of optimized hardware: moving from L0 to L3 substantially increases modeled device capability and effective QPS while driving device time per query down.

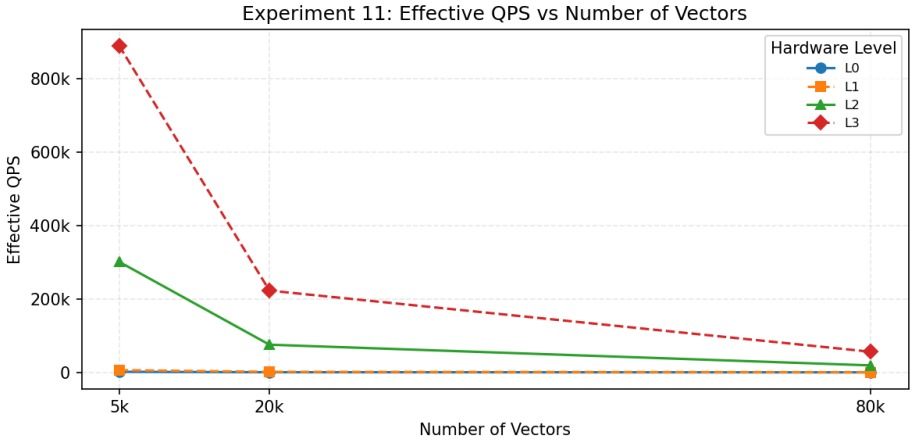

Figure 28: Experiment 11: effective QPS vs number of base vectors for hardware levels L0–L3 under approximately constant recall.

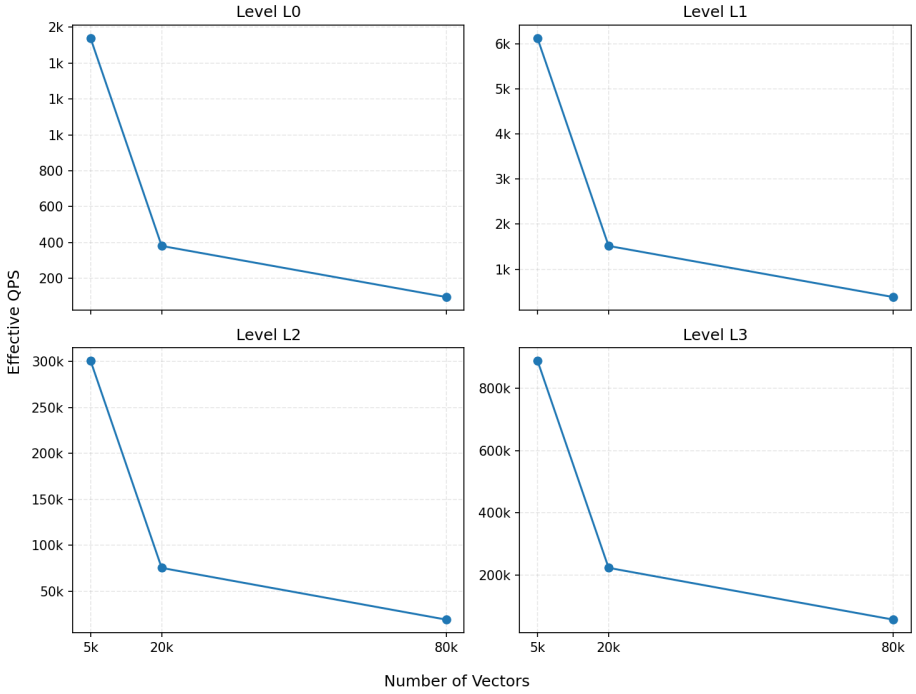

Figure 29: Experiment 11: per-level canonical effective QPS vs number of base vectors, one subplot per hardware level (faithful simulator, approximately constant recall).

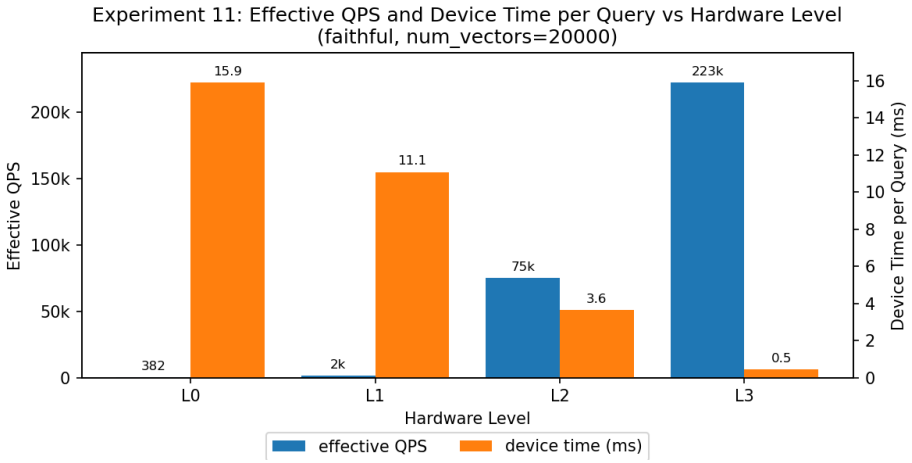

Figure 30: Experiment 11: effective QPS and device time per query (in milliseconds) vs hardware level (faithful simulator, $N_{\text{base}} = 20\text{k}$).

5.8.4 Experiment 12: Unified Comparison and Recall-Matched View

Experiment 12 combines all major solutions on common datasets and reports recall vs effective QPS (Figures 31 and 32) as well as recall-matched tables (Table 2). At moderate recall targets, DRAM and tiered designs tend to dominate, but at high recall the ANN-in-SSD configurations become competitive once their higher capacity and lower cost are taken into account. This experiment provides the clearest picture of where each design is most appropriate.

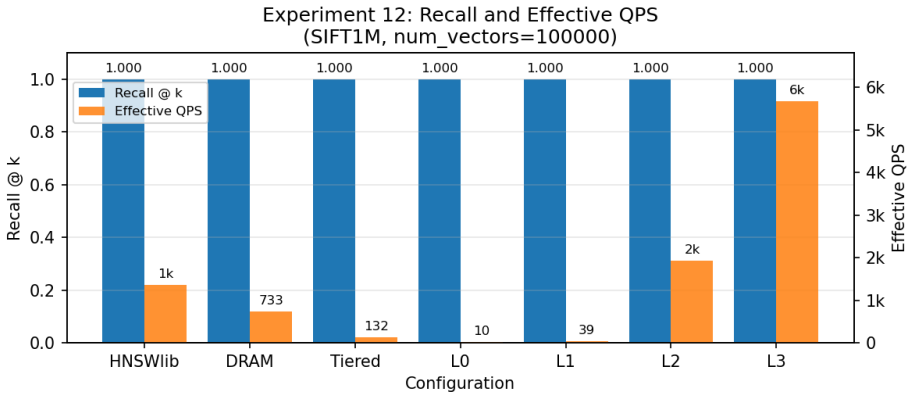

Figure 31: Experiment 12: recall @ k and effective QPS for DRAM, tiered, HNSWlib, and ANN-in-SSD configurations on SIFT1M (bar chart at the largest dataset size).

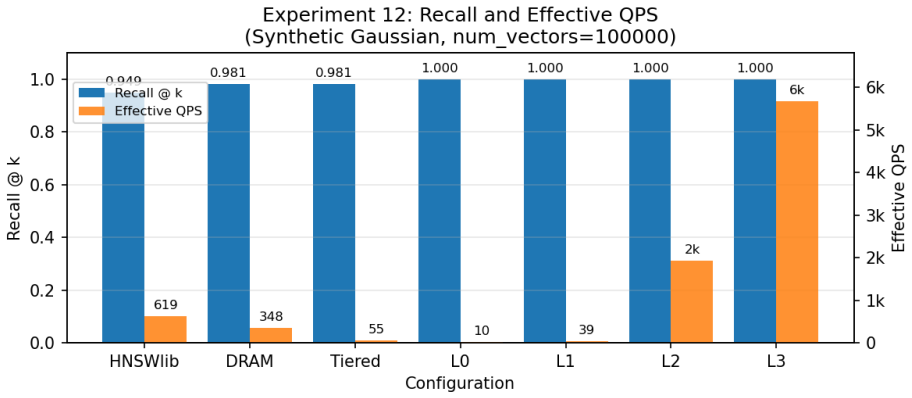

Figure 32: Experiment 12: recall @ k and effective QPS for HNSWlib, DRAM, tiered, and ANN-in-SSD configurations on the synthetic dataset (bar chart at the largest dataset size).

The recall-matched summary in Table 2 compares the best configurations for SIFT1M and the synthetic dataset at 100k base vectors. On SIFT1M at a 0.99 recall target, the DRAM and HNSWlib baselines achieve effective QPS of roughly 740 and 1,120, respectively, while an ANN-SSD L3 full-scan configuration reaches about 5,700 effective QPS with perfect recall and the ANN-SSD L2 variant still more than doubles DRAM at around 1,900 effective QPS. On the synthetic dataset at the same scale, DRAM achieves roughly 350 effective QPS at high recall while a 25% cache tiered configuration drops to about 50, yet the ANN-SSD L3 full-scan configuration maintains around 5,700 effective QPS with perfect recall—more than an order of magnitude faster than DRAM and roughly two orders of magnitude faster than the tiered design. These numbers illustrate how SSD-centric search can be attractive once media bandwidth and controller parallelism are sufficiently high.

Table 2: Recall-matched effective QPS at target recall of 0.99 for key configurations on SIFT1M and the synthetic dataset, using the recall-matched summaries from the Experiment 9/10/12 analysis scripts.

Dataset	Num vectors	Target recall	Method	Recall@k	Effective QPS
SIFT1M	100k	0.99	DRAM	1.00000	737.21
SIFT1M	100k	0.99	HNSWlib	1.00000	1120.76
SIFT1M	100k	0.99	Tiered (25% cache)	0.99950	130.88
SIFT1M	100k	0.99	ANN-SSD L2 (full scan)	1.00000	1922.24
SIFT1M	100k	0.99	ANN-SSD L3 (full scan)	1.00000	5675.11
Synthetic	100k	0.99	DRAM	0.98130	353.18
Synthetic	100k	0.99	Tiered (25% cache)	0.98075	52.74
Synthetic	100k	0.99	ANN-SSD L2 (full scan)	1.00000	1922.24
Synthetic	100k	0.99	ANN-SSD L3 (full scan)	1.00000	5675.11

Table 3: Experiment 9 recall-matched effective QPS on the synthetic dataset.

Target recall	Dataset	Method	Recall@k	Effective QPS
0.85	Synthetic	ANN-SSD L0	1.0000	48.7
0.85	Synthetic	ANN-SSD L2	1.0000	9606.2
0.85	Synthetic	ANN-SSD L3	1.0000	28370.0
0.85	Synthetic	DRAM	0.9381	3847.6
0.85	Synthetic	Tiered	0.9381	666.4

6 Conclusion

In this report we compared DRAM, tiered DRAM+SSD, and ANN-in-SSD designs for approximate nearest-neighbor search on large vector datasets. Through a series of controlled experiments we showed how cache size, cache policy, SSD characteristics, and ANN-in-SSD design parameters shape the trade-off between recall, throughput, and cost. Our results demonstrate that tiered designs let us trade DRAM capacity for modeled device time while keeping recall near the DRAM baseline, and that adjusting cache fraction directly controls I/O amplification and effective QPS. ANN-in-SSD provides additional high-recall design points that become attractive in cost-normalized terms when SSD hardware is fast and media pricing is favorable. Ignoring exact hardware and software implementation, achieving any ANN-in-SSD with performance comparable to L2 with a price reasonably comparable to current SSD technology allows for a cheap solution that can compete with HNSWlib.

6.1 Future Work

Several directions remain for future work. First, integrating the ANN-in-SSD design with more realistic storage traces or a prototype implementation would help validate the simulator assumptions. Second, exploring alternative ANN structures (e.g., product quantization or graph hybrids) within the same SSD-centric framework could yield better accuracy–latency trade-offs. Finally, extending the evaluation to larger and more diverse datasets would provide additional evidence about how these designs behave at true web scale.