Turbocharging Vector Databases using Modern SSDs

Joobo Shim
Seoul National University
jbshim@snu.ac.kr

Jaewon Oh
Seoul National University
jaewon.oh@snu.ac.kr

Hongchan Roh
Dnotitia
hongchan.roh@dnotitia.com

Jaeyoung Do*
Seoul National University
jaeyoung.do@snu.ac.kr

Sang-Won Lee
Seoul National University
swlee69@snu.ac.kr

ABSTRACT

Efficient and scalable vector search is critical for modern AI applications, particularly in retrieval-augmented generation (RAG) and large-scale semantic search. However, disk-based vector databases often suffer from significant I/O bottlenecks due to suboptimal cache hit ratios and inefficient use of modern SSD architectures. In this work, we introduce a suite of optimizations to enhance the performance of disk-resident Approximate Nearest Neighbor (ANN) indices, specifically focusing on hierarchical graph-based indexing such as HNSW. Our approach leverages three key strategies: (1) Parallel I/O leveraging io_uring to exploit SSD concurrency and reduce retrieval latency, (2) Spatially-aware insertion reordering to improve cache efficiency by dynamically adjusting insert execution order based on locality, and (3) Locality-preserving colocation to restructure index layouts and minimize costly random disk accesses.

We implement these techniques within `pgvector`, a `PostgreSQL` extension for vector search, and conduct extensive evaluations using real-world datasets. Our optimizations yield up to 11.1× improvement in query throughput, a 3.23× increase in cache hit ratio, and a 98.4% reduction in index build time. Moreover, our findings underscore the importance of SSD-aware indexing strategies for scalable vector retrieval. By integrating hardware-aware I/O optimizations with intelligent data placement techniques, this work paves the way for more efficient, high-performance disk-based vector search engines that could fully leverage modern SSD's high parallelism.

PVLDB Reference Format:
Joobo Shim, Jaewon Oh, Hongchan Roh, Jaeyong Do and Sang-Won Lee. Turbocharging Vector Databases using Modern SSDs. PVLDB, 18(11): 4710 - 4722, 2025.
doi:10.14778/3749646.3749724

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/FlashSQL/io-optimized-pgvector.

*Corresponding Author

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749724

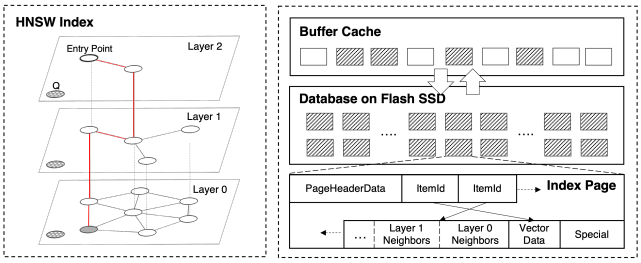
1 INTRODUCTION

Recent advancements in test-time scaling [16, 48] have significantly improved AI accuracy, pushing performance closer to near-AGI levels. This advancement, coupled with the emergence of compact yet high-performance large language models (LLMs) built on open-model frameworks [8, 15, 22, 42], is reshaping the LLM service landscape. As a result, there is increasing interest in custom data-driven Retrieval-Augmented Generation (RAG) systems and advanced retrieval techniques. Central to this transformation is dense embedding-based vector database technology, which facilitates semantic search tailored to user queries—an essential capability for modern LLM applications.

To meet the responsiveness and scalability demands of such vector databases, approximate nearest neighbor (ANN) search is widely adopted. ANN algorithms strike a balance between accuracy and efficiency by approximating exhaustive k-NN search while drastically reducing query latency. However, as the volume of vector datasets scale to hundreds of millions or more, storing vector embedding and indices entirely in DRAM becomes prohibitively expensive. In response, disk-based vector databases have emerged as an effective alternative. Systems such as DiskANN [21, 33], ScaNN [17], SPANN [11], and FreshDiskANN [39] achieve high performance by compressing indices using product quantization (PQ) or inverted file indexing (IVF)-based clustering, enabling partial in-memory storage with selective disk I/O for accessing full vectors only when necessary during search.

However, a critical limitation of such systems is that they are inherently static [45]. Their reliance on compression and clustering makes incremental updates either infeasible or highly inefficient. When updates are allowed, recall performance often degrades rapidly as the index becomes outdated [44], eventually requiring expensive periodic index rebuilds to restore quality. For example, FreshDiskANN, which allows lazy deletions, must eventually consolidate the index offline to maintain recall quality, which incurs latency spikes and operational overhead. These limitations are especially problematic for RAG and hybrid search integrated with relational databases, where updates occur continuously as new documents are ingested, modified, or removed. In such scenarios, static indices become a bottleneck and file to meet practical demands.

To support such dynamic workloads, graph-based ANN indices such as HNSW (Hierarchical Navigable Small World) [29] offer a compelling alternative due to their natural support for efficient insertions and deletions. However, adapting HNSW for disk-resident execution is nontrivial. Its greedy graph traversal, for example, requires following multiple edges across non-contiguous nodes,



(a) HNSW index search (b) HNSW index I/O process

Figure 1: HNSW index search and I/O process. (a) HNSW search traversal from the entry point to the lowest layer. The red path indicates the search sequence. (b) HNSW index storage layout, showing buffer cache usage and index page structure on flash SSD.

leading to frequent random disk I/O and poor cache locality, particularly in deeper layers of the graph where cache hit ratios are lowest. Disk-based implementations such as `pgvector` [25], despite allocating substantial buffer memory (e.g., 50% of the dataset), often suffer from these inefficiencies. The problem is compounded by the sequential and synchronous I/O model, which fails to utilize the inherent parallelism of modern SSDs, resulting in significant I/O bottlenecks. Moreover, spatially correlated queries are typically processed non-consecutively, hindering effective caching, while frequently co-accessed nodes are often stored on separate physical pages, exacerbating disk access overhead. Without careful system-level optimization, such implementations fail to deliver the performance required by large-scale vector search systems, falling short of static yet highly optimized systems like DiskANN.

To address these inefficiencies, this paper introduces a comprehensive set of optimizations designed to enhance I/O performance in disk-based vector databases employing graph-based ANN indices (such as HNSW). In this paper, our approach mainly focuses on reducing the number of costly I/O operations and improving cache hit ratios through the following strategies:

- **Exploiting SSD Parallelism via `io_uring`:** We leverage the high parallelism of modern SSDs by utilizing `io_uring` to overlap asynchronous I/O operations and enhance pipelining effects, thereby reducing latency and increasing throughput (Section 4).
- **Spatially-aware Insertion Reordering:** By reordering insert query processing to group similar insertions together, we aim to improve spatial locality and boost cache hit ratios (Section 5).
- **Locality-Preserving Colocation:** We propose pre-clustering vectors likely to be accessed together during index construction, enhancing spatial locality and reducing disk accesses in query processing (Section 6).

To our best knowledge, this work presents the first comprehensive study on I/O optimization for graph-based HNSW indices in SSD-based vector databases, providing an in-depth analysis of I/O inefficiencies and proposing a systematic approach to overcoming these challenges. By leveraging SSD parallelism, insertion reordering, and locality-preserving colocation, the proposed optimizations significantly reduce query latency and enhance throughput. An implementation of such optimization within `pgvector` demonstrates that these techniques could drastically boost the query throughput of disk-based vector databases, 11.1x improvement in read performance, a 3.23× increase in cache hit ratio, and a 98.4% reduction in indexing time. These results suggest that, with proper I/O optimizations, it is feasible to build a new class of disk-based vector databases that not only support dynamic updates but also deliver query performance comparable to highly optimized static indices such as DiskANN. Additionally, unlike static PQ-based systems that suffer from high indexing costs, our system achieves an order-of-magnitude faster index construction. Furthermore, while `pgvector` serves as the primary evaluation platform, the proposed optimizations are broadly applicable to other disk-based vector databases, highlighting their potential to enhance the scalability and efficiency of retrieval-augmented AI systems leveraging disk-based vector databases for large-scale datasets.

2 MOTIVATIONS AND OPPORTUNITIES

2.1 HNSW-based index in VectorDB

Many vector databases [1–3] utilize ANN indices, which enables fast similarity search while balancing accuracy and computational efficiency [34]. Among various ANN index alternatives, HNSW is widely used due to its robustness in handling high-dimensional data while maintaining efficient search [29]. For example, `pgvector` [25], which is an open-source `PostgreSQL` extension [37], supports vector data types and enables ANN search, including HNSW.

In this paper, we chose `pgvector` over other vector database solutions not only for its seamless integration with `PostgreSQL` but also for its dynamic index support and compatibility with the DBMS environment. Unlike static ANN systems based on PQ or clustering, `pgvector` supports incremental updates, making it well-suited for real-world applications that require both large-scale vector search and transactional integrity.

HNSW index search and construction HNSW is a hierarchical graph-based index that searches nearest neighbors efficiently by organizing data points into multiple small-world graph layers [29]. The search starts from sparsely connected top layer (Figure 1a), enabling fast traversal across distant regions. At each layer, it performs `neighbor-scan`, evaluating neighbors and retaining the most promising candidates. The process iteratively descends to lower layers until the final nearest neighbor is selected.

A key parameter, `ef_search`, controls the trade-off between accuracy and speed: higher values increase recall but require more computation. The algorithm scans neighbors leveraging spatial locality, and its efficiency depends on the graph structure. Well-connected nodes improve traversal, and high-locality queries access overlapping graph regions, enhancing cache reuse.

HNSW constructs its hierarchical graph probabilistically, keeping higher layers sparse. Insertion involves two key steps: 1) finding nearest nodes and 2) selecting neighbors to maintain navigability. This process is governed by two parameters: max neighbors per node (`m`) and candidate pool size (`ef_construction`). Higher values for both lead to better search recall but incur longer build times.

While HNSW delivers high search efficiency, its index construction is inherently slower than hash- or tree-based approaches due to search on each insertion. In `pgvector`, this issue is further exacerbated by `PostgreSQL` 's delayed deletion, which can temporarily

Table 1: Sequential vs. parallel read performance across different SSDs. The table presents the throughput of single-threaded (Seq.) vs. multi-threaded (Par.) I/O operations. All measurements were obtained using `FIO` with an 8KB request size and `ioengine=sync`, with 1 job for *Seq.* and 64 concurrent jobs for *Par.*.

Storage Device	Read MB/s Seq.	Par.	Parallel Ratio (*Par./Seq.*)	Capacity	Release Year
SSD-A†	118.0	5,606	47.5	4TB	2022
SSD-B¶	93.2	3,355	36.0	2TB	2021
SSD-C◇	121.0	2,484	20.5	1TB	2020
SSD-D*	74.1	473	6.4	512GB	2014

† Samsung PM1743, ¶ Fadu Delta, ◇ Samsung 980 pro, * Samsung 850 pro

disconnect the graph and degrade recall, requiring periodic reconstruction. Therefore, optimizing build time is critical. Similar to query execution, insertion order affects index locality. Additionally, vector layout matters: frequently co-accessed nodes should be colocated within the same pages to minimize I/O. Poor locality can cause excessive random accesses, increasing query latency.

HNSW Index I/O process Figure 1b shows that `pgvector`'s index pages follow a slotted format, where each page stores a graph node along with its vector, and neighbor tuple IDs. The graph is first built entirely in memory (`maintenance_work_mem`), enabling efficient connectivity without incurring I/O. Once the memory budget is exceeded, the index is flushed to disk, and remaining nodes are integrated incrementally. This phase involves loading pages into `buffer_cache`, potentially leading to I/O bottlenecks.

During `neighbor-scan`, neighbor pages are fetched via `buffer_cache`. Since each node has dozens of neighbors, cache misses are frequent, making search I/O-bound. Although distance calculations (e.g., L2, cosine similarity) are costly, modern CPUs handle them efficiently using SIMD and compiler optimizations [4]. Consequently, I/O efficiency—including the `buffer_cache` hit ratio—is the main bottleneck for both index construction and search. However, `pgvector` uses sequential blocked I/O during `neighbor-scan`, underutilizing SSD parallelism (see Section 2.2). Moreover, vectors are stored by insertion order, ignoring spatial locality and graph connectivity relationships. This lack of locality-aware storage results in suboptimal I/O patterns, further exacerbating random access overhead.

2.2 Parallelism in Flash SSDs

Modern flash storage architectures leverage internal parallelism at multiple levels to maximize I/O efficiency. Inside a flash SSD, NAND flash packages consist of multiple chips, each further divided into multiple planes [10, 23]. The flash controller manages these packages through multiple communication channels, allowing concurrent access to different blocks across chips [24]. This enables SSDs to handle multiple I/O requests simultaneously, significantly improving throughput when the host system issues concurrent operations.

To quantify parallelism in modern SSDs, we conducted an empirical study using four commercial drives, measuring random read throughput with FIO [6]. The test compared single-threaded sequential reads to 64-threaded random reads on a 5GB file.

Table 2: Hit ratio and SSD utilization (GloVe, 50% buffer cache)

	Cache Hit Ratio	SSD Throughput	SSD Utilization
Value	59.24%	111MB/s	1.98%

Table 3: HNSW traversal statistics (GloVe, 50% buffer cache)

Layer	Number of Nodes	Node Visits	Hit Ratio (%)
3	80	40.66	99.98
2	1,785	47.86	95.26
1	41,405	48.94	70.21
0	1,000,000	1823.57	57.49

As shown in Table 1, newer SSDs exhibit substantial internal parallelism. For example, `SSD-A` achieves 118 MB/s with one thread, but scales to 5,606 MB/s with 64 threads—a 47.5× improvement. In contrast, the older `SSD-D` shows only a 6.4× gain, limited by its legacy controller and fewer NAND channels. These results highlight the importance of exploiting SSD parallelism in high-performance storage systems, especially as newer devices adopt more channels and higher capacities.

2.3 I/O Inefficiencies of HNSW Index on SSDs

In this section, we evaluate `pgvector`'s performance and I/O inefficiencies on flash SSD using the GloVe dataset [36] (1M vectors, 200 dimensions). We built an HNSW index and executed random queries on 1% of the dataset on `SSD-A` (i.e., the highest-generation SSD model), with the buffer cache set to 50% of index size. An in-depth examination of `pgvector` 's I/O patterns, including source code analysis and empirical profiling, revealed significant inefficiencies that hinder overall performance. As shown in Table 2, SSD utilization was only 1.98%, indicating substantial underutilization of available bandwidth. Furthermore, the cache hit ratio remained at only 59.24%, highlighting inefficiencies in memory usage as well. For more details about the experimental setup, see Section 3.

Opportunity 1: Low Temporal Locality Table 2 shows that despite of a 50% buffer allocation, the cache hit ratio remains at only 59.24%, indicating inefficient cache reuse. This stands in contrast to benchmark workloads like TPC-C, where 80% of accesses typically target just 20% of the data [26].

Ideally, the HNSW index should naturally encourage temporal locality, as similar query vectors tend to traverse overlapping regions in the graph, leading to frequent reuse of cached nodes. However, a deeper examination of the execution path (as shown in Table 3), reveals a key inefficiency. Specifically, Layer 0 (the lowest layer) experiences the highest number of traversals while simultaneously exhibiting a low cache hit ratio. Since this layer is responsible for fine-grained nearest-neighbor refinement, it is the dominant factor in overall search performance and I/O overhead. In contrast, upper layers of the HNSW hierarchy contain fewer nodes and exhibit near 100% cache hit ratios, as they remain resident in memory for longer periods. This disparity suggests that the current query execution order does not maximize temporal locality, especially at lower layers. By strategically reordering query execution to increase locality-aware access patterns, cache utilization could be significantly improved, thereby reducing buffer cache misses and unnecessary disk I/O operations.

Opportunity 2: Low Spatial Locality To analyze the impact of spatial locality on pgvector 's cache efficiency and I/O performance, we measured the number of HNSW nodes accessed during a search and the corresponding index pages holding such nodes. Given that each vector has 200 dimensions (i.e., 800 bytes per vector), a single page (i.e., 8KB) can accommodate up to 10 vectors. Despite this, in our search experiment, on average, to process a single query, about 1961 HNSW nodes have to be visited by loading about 1887 index pages. In other words, the observed number of pages per not visit ratio is about 0.96, indicating that nearly every node visit results in a new page read, leading to a high cache miss rate.

This pattern suggests that, although multiple nodes are stored within the same page, they are rarely accessed together, preventing effective page reuse. Consequently increasing the frequency of page replacements in the buffer cache and amplifying I/O overhead. Ideally, the HNSW index, being graph-based, should naturally promote spatial locality, as adjacent nodes in the graph are likely to be visited together during nearest-neighbor searches. However, the current search strategy does not fully exploit this characteristic, resulting in suboptimal cache performance. Therefore, optimizing the storage layout to group frequently co-accessed nodes within the same pages could improve spatial locality.

Opportunity 3: Underutilization of SSD bandwidth Another notable observation is the significant underutilization of SSD bandwidth, as shown in Table 2, where only 1.98% of the maximum available bandwidth is achieved. This indicates a severe inefficiency in leveraging the SSD's internal parallelism, leading to suboptimal I/O performance.

The primary cause of this inefficiency lies in the sequential access pattern used during neighbor visits in the HNSW index search. Since neighbor visits in HNSW are inherently independent, they do not require a strict access order. In an optimized design, multiple SSD channels could be utilized concurrently to retrieve different parts of the index in parallel. However, the current implementation enforces a sequential access order, which significantly limits I/O parallelism. As a result, SSD operations fail to exploit modern SSD architectures capable of handling multiple concurrent requests. By restructuring the access pattern to enable parallel neighbor visits, the system could distribute I/O requests across multiple SSD channels, substantially improving bandwidth utilization and query performance.

3 EXPERIMENTAL SETUP

Hardware All experiments were conducted on an Ubuntu machine in Samsung Memory Research Center (SMRC) [14] equipped with 64GB DRAM and an Intel Xeon Gold 6442Y CPU with 24 physical cores. The system storage comprised four SSDs with distinct performance characteristics: Samsung PM1743, Fadu Delta SSD, Samsung 980 Pro, and Samsung 850 Pro. The Samsung PM1743, Fadu Delta SSD, and Samsung 980 Pro are high-performance NVMe SSDs, connected to the system via PCIe, while the Samsung 850 Pro, a SATA-based SSD, was used as a non-NVMe baseline. This configuration allowed for a comprehensive evaluation of SSD parallelism and I/O performance across different SSDs.

Table 4: Datasets used in the experiments

Dataset	Dimensions	# of Vectors	# of Queries
DBpedia	1,536	100K, 1M	10K
Deep	96	100K, 1M, 10M, 100M	10K
GloVe	200	100K	10K
NYTimes	256	100K	10K
COCO	512	100K	10K
C4	1,536	100K, 5M	10K

Software For database management, we used PostgreSQL 17 [37] with pgvector 0.8.0 [25] as the main baseline. The system was configured to support asynchronous I/O using liburing 2.9 [7], while FIO 3.38 [6] was employed for low-level I/O benchmarking. The HNSW index was constructed with ef_construction = 200 and m = 24 (See Section 2 for the detailed description of the HNSW index parameters), ensuring a balanced trade-off between index quality and construction time. The database page size was set to 8KB, a standard PostgreSQL configuration. To optimize I/O handling, io_uring depth was set to 2×m. This configuration was chosen to maximize I/O concurrency while maintaining efficient queue depth for storage interactions. Direct I/O was enabled to minimize OS-level caching, allowing accurate measurement of the direct impact of disk I/O on query performance. Any deviations from this default configuration are explicitly noted in the relevant sections.

Datasets and Evaluation Benchmark To ensure robustness and generality of our evaluation, we used a diverse set of datasets spanning various domains and characteristics, as listed in Table 4. This includes DBpedia [5], Deep [46], GloVe [36], NYTimes [31], COCO [27] and C4 [13]. For evaluation, we used ANN-Benchmarks to evaluate ANN search performance. Each dataset was randomly split into training and test sets, with the training set used exclusively for index construction and the test set for benchmarking.

4 BOOSTING I/O PARALLELISM

In pgvector, index search accesses multiple neighbor nodes to compute distances from the query vector. When these nodes are not present in the buffer cache, they must be loaded from SSD, introducing I/O latency. Since pgvector relies on PostgreSQL 's buffer management layer, data is loaded sequentially, which fails to leverage SSD parallelism effectively. Performing page access and distance calculations in a purely sequential manner significantly limits throughput as visualized in Figure 2a (pgv-orig) even though modern SSDs are optimized for parallel I/O processing.

4.1 Adapting HNSW Search to SSDs

To improve the impact of I/O parallelization, we modified pgvector to incorporate parallel I/O using io_uring, a high-performance asynchronous I/O interface in the Linux kernel. Unlike traditional I/O mechanisms such as epoll [41] and AIO [40], io_uring offers lower overhead and higher throughput by enabling batched I/O requests through shared memory queues (i.e., Submission Queue and Completion Queue) between user space and the kernel. This mechanism allows multiple read operations to be processed concurrently, maximizing SSD parallelism [12].

Adapting HNSW search to fully leverage such parallelism introduces additional challenges that go beyond simply batching read

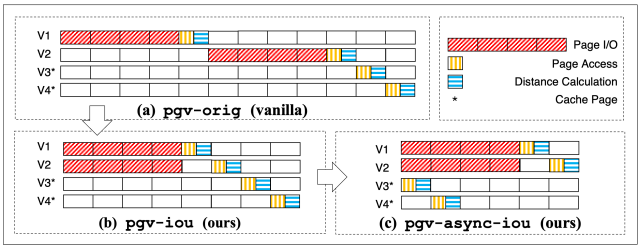


Figure 2: Comparison of query execution strategies. (a) `pgv-orig` (vanilla): Sequential query execution with high I/O overhead, (b) `pgv-iou` (ours): Parallelized I/O reduces execution time, (c) `pgv-async-iou` (ours): Asynchronous I/O further optimizes by overlapping distance calculations with I/O operations. Note that V1 and V2 undergo page I/O, while V3 and V4 (marked with asterisks) are assumed to be cached.

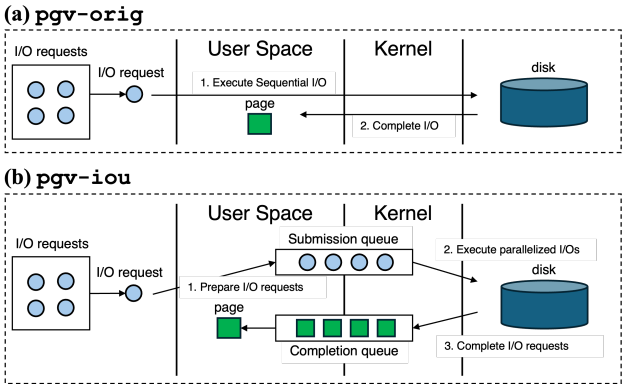


Figure 3: Comparison of I/O request flow. (a) `pgv-orig`: Sequential I/O, handling one request at a time, (b) `pgv-iou`: Parallelized I/O with submission and completion queues, enabling concurrent execution for higher efficiency.

requests. A core difficulty lies in orchestrating computation and I/O in a tightly pipelined fashion. Specifically, to avoid CPU idle time, the system must issue asynchronous reads for uncached neighbors as early as possible—before CPU-bound distance computations for cached neighbors begin. This requires careful integration of io_uring into the execution pipeline, ensuring early I/O issuance and seamless overlap between compute and I/O phases.

Further complexity arises from the asymmetric distribution of pages across SSD channels. If the system waits for all outstanding I/O requests to complete, CPU resources may remain underutilized. To address this, we adopt a responsive polling strategy that triggers distance computation as soon as any page becomes available, avoiding full-batch synchronization delays. Additionally, we tune the min_complete parameter in io_uring to balance I/O responsiveness and CPU overhead. Empirical tuning reveals that a moderate setting (e.g., min_complete = 6) achieves robust performance across diverse workloads.

We applied I/O parallelization to the `neighbor-scan`, which runs to compute distances between the query node and its neighbors. The neighbor list serves as the candidate set for I/O operations, excluding already visited nodes and cached pages. Then asynchronous read requests are added to the Submission Queue. Once the batch of requests is prepared, parallel read operations are initiated. Upon

Algorithm 1 Asynchronous Parallel Neighbor Retrieval

```
1:  Q: Query Vector
2:  N: Unvisited Neighbor nodes
3:  cached ← {n ∈ N | n ∈ buffer_cache}          ▷ Cached neighbors
4:  uncached ← N \ cached                        ▷ Uncached neighbors
5:  function EvaluateNeighbors
6:      Submit uncached to io_uring
7:      for c ∈ cached do
8:          Compute Distance(Q, c)
9:      end for
10:     while uncached ≠ ∅ do
11:         r ← wait for any I/O completion
12:         Compute Distance(Q, r)
13:         Remove r from uncached
14:     end while
15: end function
```

completion, retrieved pages are placed in the Completion Queue and made available to the search process only after all outstanding I/O requests have been finalized. An example of the scan process of such improvement is shown in Figure 2b (`pgv-iou`), and the I/O request flows of `pgv-orig` and `pgv-iou` are depicted in Figure 3.

In `pgv-iou`, I/O overlapping significantly reduces overall execution time compared to `pgv-orig`. However, this also shifts the relative time distribution, increasing the proportion spent on page access and distance calculation. Since page I/O is handled by the storage layer and distance calculations by the CPU, these two processes can theoretically be executed in parallel. Despite this, even in `pgv-iou`, the CPU remains idle while waiting for I/O operations to complete, introducing performance inefficiencies.

To address this issue, we implemented pipelined distance calculation, a technique that dynamically overlaps I/O operations with computation. Instead of treating I/O and distance computation as distinct sequential phases, this approach enables concurrent execution, ensuring that CPU cycles are utilized efficiently while I/O operations are in progress as visualized in Figure 2c (`pgv-async-iou`).

More details are demonstrated in Algorithm 1. The system immediately processes cached pages upon submitting I/O requests, minimizing CPU idle time (LN 8). Additionally, rather than waiting for all I/O requests to complete, the system processes pages as soon as any I/O request is finalized (LN 12). This is achieved through continuous monitoring of the `io_uring` submission queue, using a real-time polling mechanism known as peeking. By detecting completed I/O requests dynamically, the system avoids unnecessary blocking and schedules distance calculations opportunistically, maximizing CPU utilization.

4.2 Evaluation

To assess the impact of I/O parallelization in `pgvector`, we measured both search performance (in queries per second, QPS) and index creation efficiency (in elapsed time) using the DBpedia-1M dataset. The HNSW index was configured with `ef_search` = 40. Experimental results demonstrated that search performance improved by up to 8.55×, while index creation time was reduced by 85.07%, highlighting the significant performance gains achieved through parallelization.

The traditional read approach struggles to fully utilize SSD bandwidth due to CPU overhead from frequent context switching and system calls, even at high concurrency levels. In contrast, `io_uring` minimizes these inefficiencies by leveraging a ring buffer to batch

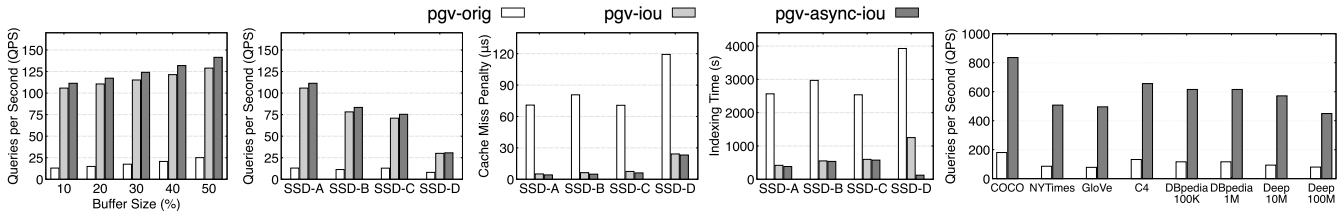


Figure 4: Effect of I/O parallelization across different SSD architectures and datasets. (a) Query performance with varying buffer sizes on SSD-A, (b) Performance comparison across SSDs, (c) Cache miss penalty reduction through parallel I/O, (d) Improved index construction speed with I/O parallelization, (e) Impact of high concurrency on various datasets. Note that (a–d) results are based on the DBpedia-1M dataset.

and process I/O requests asynchronously, significantly reducing system call overhead and context switching costs. To validate this, we compared pgv-orig and pgv-async-iou under a 10% buffer condition, measuring SSD read bandwidth as concurrency increased until performance saturation. In our tests, pgv-orig reached 7,984MB/s at a concurrency level of 200, whereas pgv-async-iou achieved 8,902MB/s with only 30. This demonstrates that io_uring enables higher SSD utilization with lower concurrency requirements, effectively preventing the performance saturation observed in traditional methods.

Effect of Buffer Size on Query Performance Figure 4a presents the effect of I/O parallelization on query throughput per second (QPS) as the buffer size increased from 10% to 50% on SSD-A. The largest performance difference was observed in 10% buffer size, where the original pgvector (pgv-orig) recorded 13.02 QPS, while pgv-iou reached 105.81 QPS (an 8.12× improvement), and pgv-async-iou further improved to 111.4 QPS (an 8.55× increase). The gap between pgv-iou and pgv-async-iou was most pronounced when the buffer size was 50%, with the Pipeline method delivering an additional 10% performance improvement due to its overlapping execution strategy.

As the buffer cache size increased, all approaches exhibited consistent QPS improvements. However, the relative performance gain of pgv-iou over pgv-orig diminished (e.g., from 8.12× with 10% buffer to 5.14× with 50% buffer). This trend occurs because a larger buffer size increases the hit ratio, reducing the number of I/O requests via io_uring, thereby limiting the effectiveness of SSD parallelism. In contrast, the Pipeline method became more effective as the buffer cache grew, since a larger cache allowed for more pages to be preprocessed before the first I/O request was completed.

Effect of SSD Parallelism on Query Performance Figure 4b compares the impact of SSD parallelism on query performance across different SSD models. Despite variations in maximum read speeds, pgv-orig showed minimal performance variation, as it does not leverage SSD parallelism, resulting in low SSD utilization. In contrast, higher level of I/O parallelization showed greater benefits on SSDs with higher internal parallelism. For example, on SSD-D (lowest parallelism) performance improved by 3.82× compared to pgv-orig, whereas on SSD-A (highest parallelism) performance increased by 8.55×. Figure 4c illustrates the cache miss penalty (i.e., page access latency) across different SSDs. As shown in the figure, I/O parallelization significantly reduced average I/O latency during cache misses, leading to lower penalty costs. With pgv-async-iou, the cache miss penalty reduction compared to pgv-orig was 80.5% on SSD-D and 94.1% on SSD-A, demonstrating that higher SSD parallelism amplifies the benefits of parallelized I/Os. These results indicate that as SSD architectures continue evolving, newer models with superior parallelism will further enhance the effectiveness of I/O parallelization, making them increasingly valuable in modern vector databases.

Effect of I/O Parallelization on Index Construction In vector databases handling massive datasets, index construction is just as critical as search performance, particularly in environments that require continuous data updates. I/O parallelization also improves index creation speed by reducing the time spent in neighbor-scan, which are I/O-intensive operations. To quantify this effect, we measured the time required to insert an additional 1% of data into an existing index that had already been built with 99% of 1M DBpedia vectors. As shown in Figure 4d, reducing I/O wait times significantly accelerated index updates, improving the efficiency of large-scale vector processing. For example, pgv-async-iou completed the insertion in 383.04 seconds on SSD-A, an 85.07% reduction compared to pgv-orig, which required 2,566.29 seconds.

Concurrency, Scalability, and Dataset Diversity Figure 4e shows the performance across diverse datasets under concurrency 10 using SSD-A. The datasets vary in size and dimensionality: DBpedia (1,536 dim, 100K/1M), Deep (96 dim, 10M/100M), NYTimes (256 dim, 100K), GloVe (200 dim, 100K), COCO (512 dim, 100K) and C4 (1,536 dim, 5M). I/O parallelization consistently improved performance, with gains ranging from 4.63× (COCO) to 6.34× (GloVe)—levels that pgv-orig would require 60–70 concurrency to match. For DBpedia-1M, the gain dropped from 8.55× (single-threaded) to 5.29×. This decline is attributed to performance degradation caused by page locking and other contention within the shared buffer among multiple threads.

5 SPATIALLY-AWARE INSERTION REORDERING

As discussed in Section 2.3, executing queries in an arbitrary order leads to inefficient cache utilization, as the cache state for each query is influenced by the execution sequence of preceding queries. The execution order directly affects the degree of overlapping traversal paths, which in turn impacts cache hit rates. To quantify

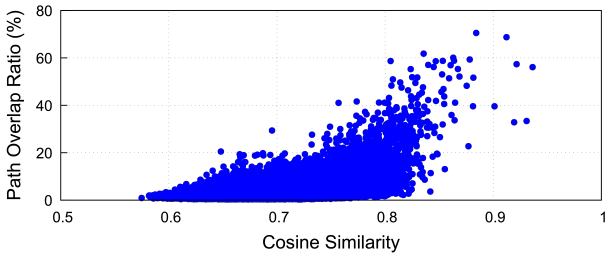


Figure 5: Correlation between vector similarity and overlapping search paths. Higher semantic similarity leads to greater path overlap, reinforcing the effectiveness of spatially-aware insertion reordering.

this effect, we analyze the relationship between vector distance and path overlap. Specifically, we randomly selected 500,000 pairs from the DBpedia dataset and computed their cosine similarity. The path overlap ratio represents the proportion of nodes visited by a later query that were also traversed by an earlier query. Figure 5 illustrates this relationship by depicting the correlation between vector pairwise distances and the number of overlapping nodes encountered at the search. Even when two queries are not identical, they frequently traverse shared neighbors or similar regions of the graph, enhancing spatial locality and improving cache efficiency.

To address this issue, we implement spatially-aware insertion reordering, an optimization technique that adjusts insert query execution order to maximize page reuse. By ensuring that upcoming queries are spatially closer to previous ones, this approach reduces redundant page loads, improves cache efficiency, and therefore minimizes I/O overhead.

5.1 Effectiveness of Insertion Reordering

Figure 6 presents the reordering time (log scale) and cache hit ratio improvements for various reordering approaches. The red diamond markers denote projection-based reordering techniques, including Random Projection [18], Centroid-Based Reordering (based on dataset centroid), First-Element Reordering (based on the first element of each vector), and PCA-Based Reordering utilizing Randomized SVD [9]. The blue circle markers indicate cluster-based reordering methods, such as K-means Clustering [28], Gaussian Mixture Model Clustering [38], Spectral Clustering [32], and Hierarchical Clustering [35].

The experimental results reveal a fundamental trade-off between reordering strategies. Projection-based reordering achieves lower computational overhead, enabling faster reordering times. In contrast, cluster-based reordering leverages the intrinsic structure of the data, leading to improved spatial locality and higher cache hit ratios. The key rationale behind this difference is that cluster-based reordering groups semantically similar vectors together, making query access patterns more spatially aware, thereby enhancing cache efficiency.

Among projection-based approaches, PCA-based reordering exhibited the highest cache hit ratio, as it effectively reduces dimensionality while preserving important data variance. Meanwhile, among cluster-based methods, K-means clustering achieved the fastest reordering speed, striking a balance between computational efficiency and spatial awareness in query execution.

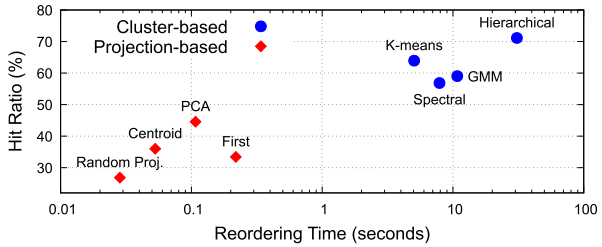


Figure 6: Performance impact of embedding reordering strategies. Comparison of projection-based (red, diamond) and cluster-based (blue, circle) reordering techniques in terms of cache hit ratio and preprocessing time. Note that the cache hit ratio of `pgv-orig` without reordering was 22.36%

5.2 Index Building with Insertion Reordering

The most time-consuming aspect of index construction is page I/O operations, which occur when traversing partially constructed graph layers to locate the nearest existing nodes. As shown in Figure 6, reordering insert queries before indexing improves buffer hit ratio by enhancing spatial locality and minimizes expensive page accesses. In scenarios where data is primarily appended, rebuilding the entire index from scratch is computationally prohibitive. Therefore, in such cases (e.g., LN 1 of Algorithm 2), only newly inserted data is reordered, leaving the existing index structure intact. This approach ensures that newly added vectors are organized to improve locality and index efficiency.

The trade-off between reordering speed and cache hit ratio must be carefully managed when selecting an optimal reordering strategy. For smaller datasets, more fine-grained reordering techniques can be employed to maximize spatial locality. However, for larger datasets, the computational cost of reordering increases significantly, making it crucial to adjust the reordering granularity to balance efficiency and performance.

5.3 Evaluation

To evaluate the impact of reordering on index construction, we examined two reordering strategies: PCA reordering (Projection-based) and K-means reordering (clustering-based). The experiments were conducted under two indexing scenarios. In the full reordering scenario, the entire dataset was reordered before insertions. In the incremental reordering scenario, only the newly inserted portion was reordered before incorporating it into the pre-existing index. In this way, even without knowing the full workload in advance, reordering can still effectively reduce I/O and accelerate index construction.

All experiments were conducted on `SSD-A`, with the buffer cache size set to 20% of the dataset size. 95% of the dataset was pre-indexed, and the remaining 5% was inserted incrementally. For K-means reordering, data was clustered in chunks of 10,000 vectors to maintain linear computational complexity, with the number of clusters set to 10. After reordering the vectors accordingly, the index was constructed, and the performance of both scenarios was compared and analyzed.

Figure 7a shows that reordering significantly reduces index construction time while improving buffer cache locality, even when considering the additional preprocessing overhead. For instance,

Algorithm 2 Insertion with multiple insert pages

Require: Nodes to insert N, Multiple Insert Pages $\mathcal{I}$, Fallback Insert Page $\mathcal{F}$
Ensure: All nodes in N are inserted in an appropriate page

$N' \leftarrow$ sort N `using spatially-aware insertion reordering`
for each node $n \in N'$ **do**
  $C \leftarrow \emptyset$ ▷ Candidate partitions
  for each partition $p \in \mathcal{P}$ **do**
    $count \leftarrow$ count_neighbor_overlap(n, p)
    $C \leftarrow C \cup (p, count)$
  end for
  Sort C in descending order of count
  for each $(P, count)$ in C **do**
    if $P \in \mathcal{I}$ **then**
      $page \leftarrow$ find_available_page(P)
      if is_page_full$(page)$ **then**
        $page \leftarrow$ create_extended_page(P)
      end if
      insert_into_page$(n, page)$
      return
    end if
  end for
  insert_into_page$(n, \mathcal{F})$ ▷ If no valid partition exists, use fallback insert page
end for

PCA and K-means reduced index build time by 68.2% and 67.5% when full indexing, respectively, compared to `pgv-orig`. PCA required only 10.39 seconds for reordering, whereas K-means reordering took 510.97 seconds. However, K-means achieved a slightly higher buffer cache hit ratio (86.62%) compared to PCA (86.58%), both significantly improving over `pgv-orig` (53.37%). For incremental indexing, K-means reduced build time by 67.3%, while PCA achieved a 42% reduction. Reordering times remained low, at 0.588 seconds for PCA and 25.2 seconds for K-means, having little effect on total indexing time. The buffer cache hit ratio reached 74.35% for PCA and 86.58% for K-means, indicating that clustering-based reordering preserved better cache efficiency.

A higher buffer cache hit ratio reduces buffer eviction frequency, lowering the amount of data written to disk. Since SSDs have asymmetric read/write performance, where writes are slower than reads, a reduction in write volume directly contributes to shorter index build times. Additionally, minimizing write operations extends SSD lifespan, reducing long-term storage costs. Figure 7b shows the write volume for each indexing scenario. Higher cache hit ratios and shorter build times correlated with lower total write volume. The greatest reduction was observed in the PCA with full reordering case, where total write volume decreased by 46.4% compared to `pgv-orig`. This improvement is attributed to increased page reuse, which minimizes redundant disk writes.

While reordering improved buffer cache hit ratio and reduced indexing time, a significant portion of indexing time was still spent on I/O operations. To further reduce indexing overhead, we applied I/O parallelization (i.e., `pgv-async-iou`) alongside reordering and measured its impact on index build time. As shown in Figure 7c, applying PCA under `pgv-async-iou` resulted in a 90.3% reduction in index build time, completing the indexing process in 1,226 seconds, including reordering time. Similarly, K-means under `pgv-async-iou` reduced build time by 84.9%, completing in 1,915 seconds. These results demonstrate that combining reordering with I/O parallelization accelerates index construction by more than 10× compared to the vanilla approach, making it a highly effective optimization strategy for large-scale vector indexing. Figure 7d presents an evaluation of the scalability and effectiveness of insertion reordering in reducing index construction time (including sorting time) under concurrent ingestion with five parallel threads across multiple datasets. In each dataset, 95% was pre-indexed, and 5% was newly inserted. For large datasets (Deep-10M, Deep-100M, and C4), 50,000 vectors were inserted. The results show 8.05× to 13.82× reduction in indexing time, with the best performance when full sorting was applied. In addition, we examined whether spatially-aware insertion reordering affects the structure of the graph and, consequently, query throughput. However, it had no meaningful impact in the graph structure itself. The sorting did not lead to any explicit colocation effects, and its influence on cache hit ratio and query performance was negligible.

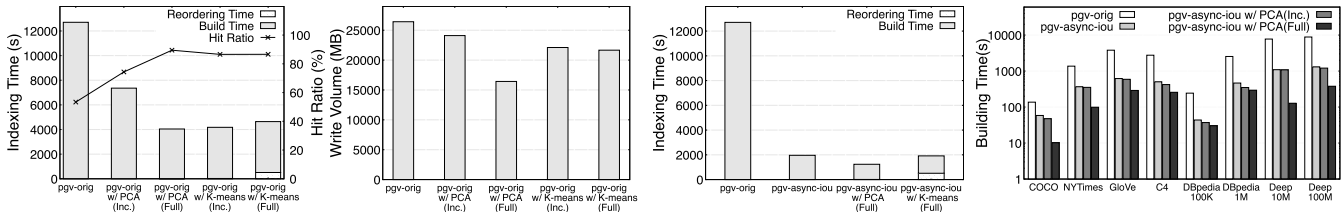
6 LOCALITY-PRESERVING COLOCATION

In disk-based vector search, maintaining spatial locality on disk is critical for reducing disk I/O. Traditional database indexes, such as B+ trees, inherently preserve locality by sorting data keys, ensuring that adjacent keys are stored within the same disk pages. However, graph-based vector indexes, such as HNSW, rely on nearest-neighbor relationships rather than a fixed ordering, so even pre-reordering cannot ensure that graph neighbors will be physically stored together.

To address this challenge for preserving locality when storing vectors in disk-based indexes, we employ Block Neighbor Frequency (BNF)-based partitioning during initial index construction [43]. BNF iteratively assigns each node to the page that contains most of its neighbors, ensuring that strongly connected nodes are colocated. In its standard form, BNF operates at the page level, where each page contains a fixed number of nodes. To improve flexibility, we extend BNF by introducing partition-based computation, allowing the number of nodes per partition to be dynamically adjusted. This generalized approach preserves the core principles of BNF while providing greater adaptability in organizing and storing data.

In addition, while the original BNF-based partitioning improves locality during initial index construction, it does not account for incremental insertions. When new vectors are appended without considering the existing partition structure, locality degrades over time, leading to increased random disk accesses. This issue is particularly problematic when datasets exceed memory limits or vectors are frequently updated, disrupting the original data organization.

In disk-based HNSW implementations such as `pgvector`, newly inserted nodes are sequentially appended to a designated insert page after the initial index is flushed to disk, with no regard for graph connectivity. This leads to several structural inefficiencies. First, inserted nodes are stored in a single page rather than near their graph neighbors, increasing random access costs. Second, unlike B+ trees, which split nodes upon insertion to maintain locality, HNSW does not dynamically adjust partitions. Once a page is written to disk, modifying its structure requires extensive rewrites, which are impractical. Third, as more insertions occur, neighbors become increasingly distributed across multiple pages, further increasing random disk accesses and degrading retrieval performance. Thus, a locality-aware insertion strategy is required to preserve locality on disk without necessitating full re-partitioning. By ensuring that newly inserted nodes are physically placed in proximity to their graph neighbors, disk-based vector search can maintain high retrieval efficiency while minimizing I/O overhead.



(a) (b) (c) (d)

Figure 7: Influence of reordering on index construction. (a) Indexing time and cache hit ratio, (b) Total write volume during index construction (c) Acceleration of index construction, (d) Indexing time with various datasets under high-concurrency

6.1 Insertion Strategy

To mitigate locality degradation caused by insertions, we introduce a locality-aware insertion strategy (as described in Algorithm 2) that dynamically selects insertion pages based on graph structure. Unlike conventional approaches that append all new nodes to a single insert page, our method assigns insert pages to specific partitions, ensuring that newly inserted nodes remain close to their nearest neighbors in the graph. When inserting a new node, the strategy first analyzes its neighboring nodes to determine the most suitable partition (LN 5-6). If the selected partition already has an insert page, the node is placed there (LN 15). If the insert page is full, a new page is allocated and linked to the partition (LN 12-14). If no partition-based insert page is available, the node is stored in a fallback insert page, preventing excessive fragmentation (LN 19). Additionally, when inserting new data, we can apply the insertion reordering strategy from Section 5 to improve spatial locality during insertion (LN 1). This serves as an orthogonal optimization that complements our method, which focuses on preserving locality during retrieval.

To ensure efficient insert page management, we dynamically adjust the proportion of partitions using insert pages, preventing excessive page expansion while maintaining query locality. This approach extends BNF principles beyond initial index construction, applying them to incremental insertions without the computational overhead of full re-partitioning. By dynamically distributing new data based on graph connectivity, our method ensures that disk-based HNSW maintains spatial locality, supporting efficient query performance even as new data is continuously inserted.

In our implementation within `pgvector`, insertions are managed through a Partition Page structure which is newly introduced to maintain a mapping of partition IDs to their corresponding insert page numbers. Since each 8KB Partition Page can store roughly 1,000 partition-to-insert page mappings, it supports large-scale indexing without excessive memory usage. During an insertion, Partition Pages are first loaded to retrieve the mapping between partitions and their associated insert pages. If a new insert page is allocated for a partition, the Partition Page gets updated by recording the newly assigned page number.

The computational complexity of the multiple insert pages approach is analyzed as follows $O(P \cdot o \cdot |V|)$, where P is the number of insert pages, o is the average out-degree of nodes, $|V|$ is the number of inserted nodes. Since most operations involve simple lookups rather than complex graph traversals or distance computations, disk I/O remains the primary bottleneck in insertion rather than CPU processing. Each insertion involves reading all partition pages to locate the correct insert page, and only one of them is updated if necessary. Since these pages store only metadata and are relatively small, they are likely to be cached, reducing disk accesses. Writes are infrequent, as new insert pages are allocated only when needed, keeping I/O overhead low. As the number of insert pages increases, the total number of index pages grows accordingly. Each insert page may also contain unused space, leading to potential inefficiencies in storage utilization. However, our locality-aware insertion reduces fragmentation by keeping related data clustered.

We note that deletion is handled via PostgreSQL's standard vacuum mechanism. Deleted nodes are first marked in the heap and later physically removed by vacuum, during which the graph edges referencing them are also updated to maintain HNSW connectivity. The resulting freed index slots can be reused by subsequent insertions. While our system does not explicitly reassign these slots based on partition locality, fallback insertions that overwrite freed space may still help preserve spatial locality after deletions.

6.2 Evaluation

In this section, we evaluate the impact of partitioning strategies and locality-aware insertions on query performance and storage efficiency. We first analyze how varying the number of nodes per partition and page sizes affects hit ratio and query throughput. We then examine how locality-aware insertion maintains efficiency as new data is added to the index.

Impact of Partitioning Figure 8a analyzes how partitioning influences hit ratio by varying the number of nodes per partition. The vanilla setting in `pgv-orig`, where each partition contains only one node, is incrementally increased to assess the impact on locality and cache efficiency. Results show that hit ratio improves as more nodes are stored together, but the effect saturates at 64 nodes per partition across datasets, with diminishing gains beyond this threshold. However, the maximum hit ratio gains vary across datasets. For example, in the Deep dataset, hit ratio increases from 15.84% to 51.19% (3.23×), while in COCO, it rises from 22.1% to 41.02% (1.86×). However, datasets like NYTimes and GloVe exhibit lower gains, with NYTimes improving from 13.25% to 18.71% (1.41×) and GloVe from 13.99% to 27.95% (2.00×). This variation could be explained by two possible factors: (1) HNSW clustering coefficient, which quantifies how well a node's neighbors are interconnected, and (2) nodes per page, which determines how many nodes are stored together within a single disk page. For instance, datasets like NYTimes and GloVe, with relatively low clustering coefficients of

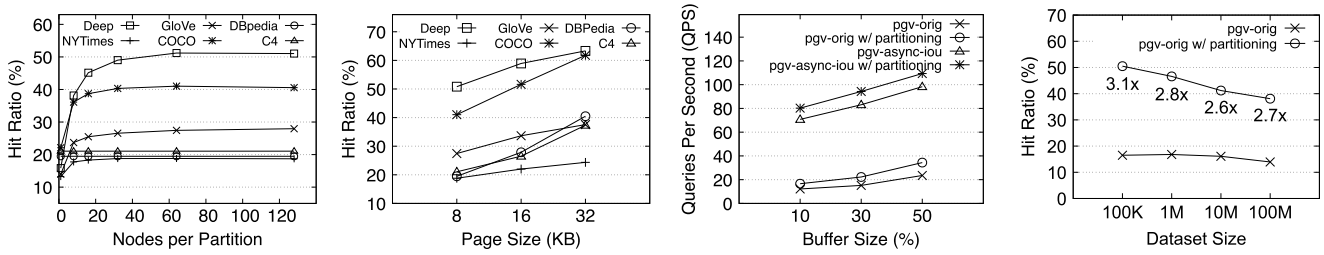


Figure 8: Impact of partitioning on hit ratio and query throughput. (a) Impact of the number of nodes belonging to one partition on hit ratio, measured using various datasets of 100K vectors, (b) Impact of page size on hit ratio, measured using the same datasets as in (a), (c) Impact of buffer cache size on query throughput, measured using the DBpedia 100K dataset with a 32KB page size, (d) Impact of dataset size on cache hit ratio, measured using the Deep dataset with sizes ranging from 100K to 100M.

0.02, require more search hops, limiting partitioning effectiveness. In contrast, COCO and Deep, which exhibit stronger connectivity at 0.08 and 0.05, respectively, improve locality and reduce disk accesses. Furthermore, a larger number of nodes per page enhances locality by minimizing disk I/O. Deep-100K benefits from storing 10 nodes per page, whereas NYTimes and GloVe, with only 7 and 6 nodes per page, respectively.

Figure 8b further shows that larger page sizes improve hit ratio, as fewer I/O operations are required when more nodes fit within a single page. To assess the combined impact of partitioning and I/O parallelism, we evaluate the DBpedia-100K dataset. Figure 8c demonstrates that partitioning alone improves data locality, while integrating `pgv-async-iou` further accelerates query throughput by reducing disk bottlenecks. This effect is particularly pronounced when buffer size is limited. At 10% buffer capacity, for example, query throughput improves significantly (i.e., 6.6×) compared to the vanilla setting in `pgv-orig`. To evaluate scalability, we extended our experiments to datasets with up to 100 million vectors. Figure 8d shows how the hit ratio changes as the dataset size increases. Although the absolute gain slightly decreases due to increased graph sparsity, partitioning still achieves a meaningful 2.7× improvement over `pgv-orig` at 100M, confirming its effectiveness at scale.

Impact of Locality-aware Insertion Next, we evaluate locality-aware insertion under a scenario where 90% of the dataset is pre-indexed, and the remaining 10% is inserted incrementally. As more partitions adopt locality-aware insertions, additional insert pages are allocated, leading to index page growth.

Figure 9a shows that when insert pages are assigned to 90% of partitions, the total number of index pages increases by approximately 7%. Despite this increase in index size, insertion performance remains stable. More interestingly, as Figure 9b illustrates, insertion throughput continues to improve despite the minor computational overhead introduced by `locality-aware insertion` (i.e., Lines 4–18 in Algorithm 2), which involves lightweight operations such as neighbor-overlap counting, candidate partition sorting, and page availability checks, rather than expensive disk I/O or memory copying. For instance, in disk I/O-bound settings, the measured overhead across all datasets is negligible, ranging from 0.05% to 0.1% of total insertion time. Even in the worst-case in-memory scenario, the overhead remains below 1%. This minimal cost is outweighed by the benefits of enhanced data locality and cache-friendly partitioning, which boost the buffer cache hit ratio and ultimately result in a net performance gain. Figure 9c further supports this by showing that increasing the number of insert pages substantially improves the buffer cache hit ratio, thereby reducing page access latency.

As the insertion ratio increases, maintaining locality becomes more challenging, since newly inserted nodes are increasingly dispersed across multiple pages. This issue becomes more pronounced as the proportion of partitions using insert pages grows, amplifying the benefits of locality-aware insertion. Figure 9d illustrates that locality-aware insertion maintains a higher hit ratio, while vanilla insertions in `pgv-orig` experience a steady decline. When only 10% of the index is pre-built, and 90% of data is inserted incrementally, for example, the hit ratio in the vanilla setting in `pgv-orig` continuously declines, reaching 19.14%. In contrast, locality-aware insertion sustains a significantly higher hit ratio of 44.16%—approximately 2.31× higher than vanilla, demonstrating its effectiveness in preserving spatial locality over time.

As shown in Figure 10, the integration of our proposed techniques (i.e., `pgv-ours`) leads to a substantial improvement in data locality. In particular, the holistic insertion method achieves up to 7.6× higher insertion throughput compared to `pgv-orig`, while improving the insert hit ratio from 43.5% to 79.1%. These findings indicate that combining pre-insertion sorting with locality-aware placement accelerates ingestion (with improvements in retrieval performance discussed in Section 7.) It is worth noting that DiskANN does not support incremental insertions and is therefore excluded from this comparison; its end-to-end index build time from scratch is analyzed separately in Section 7.

7 QUANTITATIVE COMPARISON WITH `DISKANN`

In this section, we quantitatively compare our approach with `DiskANN`, a state-of-the-art disk-based ANN systems, focusing on search performance and index build time. Importantly, the two systems differ fundamentally in architecture: `DiskANN` loads a PQ-compressed vectors into DRAM and accesses full vectors from storage, while the HNSW-based `pgvector` operates directly on full-precision vectors using DRAM as a shared buffer. As a result, `DiskANN` and similar systems like SPANN and ScaNN are optimized for static data and lack efficient update support. In contrast,

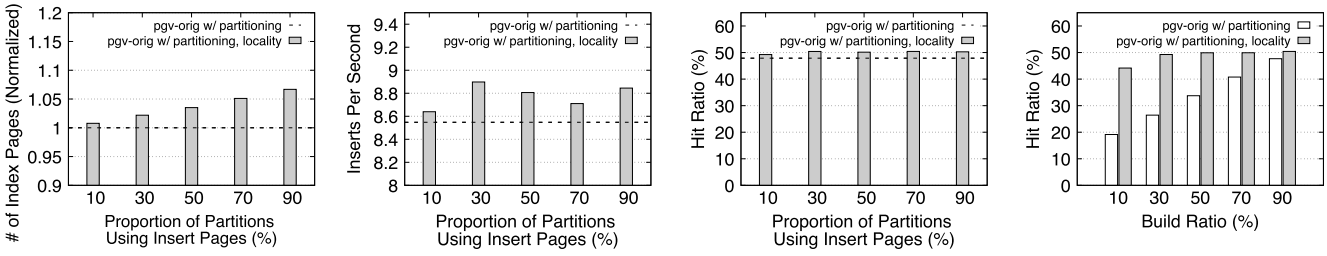


Figure 9: Impact of locality-aware insertions (i.e., `w/ partitioning, locality`) on indexing efficiency and search performance, measured using the Deep 100K dataset. (a) Growth in the number of normalized index pages with increasing partitions using insert pages, (b) Stable insertion throughput despite index growth, (c) Comparable buffer cache hit ratio due to optimized insert locality, (d) Long-term preservation of spatial locality in the index

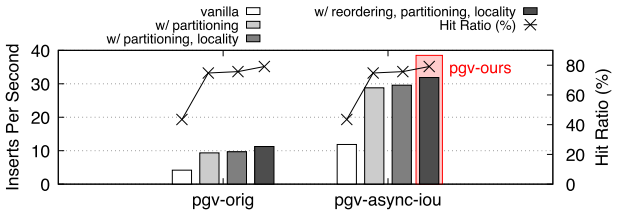


Figure 10: Impact of individual strategies in our holistic approach (i.e., `pgv-ours`) on the Deep-100K dataset. Each strategy is applied during the incremental insertion of 10% of the dataset. Results are shown for both `pgv-orig` and `pgv-async-iou`.

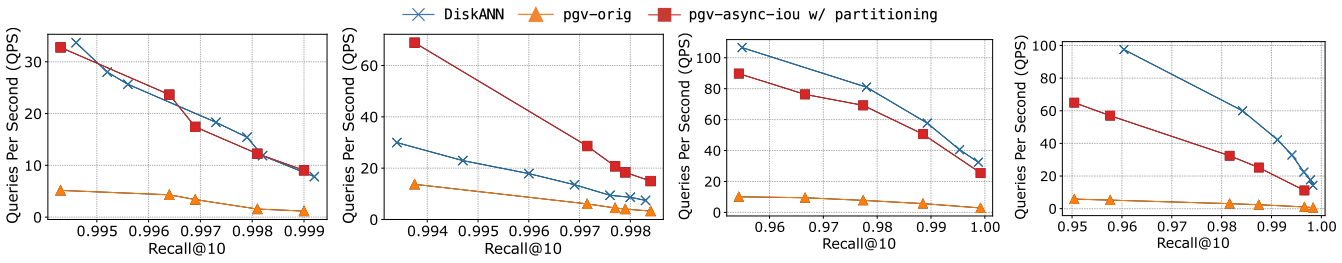
`pgvector` delivers comparable search performance with the added benefit of efficient updates. Under cache-friendly workloads, it even surpasses `DiskANN` in query throughput. Moreover, the index build time differs by approximately an order of magnitude, with `pgvector` being significantly faster.

Search Performance Comparison To ensure a fair comparison, we configured `DiskANN` with parameters aligned to our approach: `max_degree` was set to 48 (matching the number of neighbors in HNSW Layer 0), and `Lbuild` to 200 (analogous to `ef_construction`). During search, `pgvector` used a buffer size of 10% of the dataset, while `DiskANN` was given unlimited memory via `search_DRAM_budget`. Beam width was set to 4, following the original `DiskANN` paper. For evaluation, we used the C4 dataset. Alongside randomly sampled queries, we included queries from the MMLU benchmark [19]—a widely used dataset for LLM and RAG evaluation. MMLU contains over 17,000 questions across diverse, thematically grouped domains, making it a realistic, cache-friendly workload for assessing performance under practical conditions.

As shown in Figure 11a, the baseline `pgv-orig` falls far behind `DiskANN` on randomly selected queries, whereas `pgv-async-iou` achieves comparable performance. Notably, `pgvector`-based methods improve with larger buffer sizes, as the low cache hit ratio (<20%) leaves room for further gains. In contrast, `DiskANN` cannot benefit from additional memory due to limits for efficiency in its PQ representation, which is capped at 512 bytes per vector or 1 byte per dimension [30]. Figure 11b shows results for the MMLU query set, where high inter-query locality leads to a cache hit ratio exceeding 80%. In this cache-friendly scenario, `pgv-async-iou` significantly outperforms both `DiskANN` and `pgv-orig`, highlighting its ability to leverage shared buffer reuse—something `DiskANN` is not designed to exploit. On the Deep-1M dataset (Figure 11c) and the Deep-100M dataset (Figure 11d), `DiskANN` outperforms `pgvector`-based systems overall. However, our optimized variant (`pgv-async-iou` with partitioning) significantly narrows this performance gap. For example, on Deep-100M, it achieves an 11.1× speedup over `pgv-orig` and reduces the original 19.1× gap with `DiskANN`. These improvements stem from I/O optimization through parallelization and an enhanced cache hit ratio enabled by locality-preserving colocation, demonstrating strong scalability even under large-scale, low-locality workloads.

In summary, our system supports mutable indexes and SQL integration—capabilities absent in `DiskANN` —while delivering competitive or superior performance across varying workloads. Moreover, `pgvector`-based search benefits from increased memory and locality, making it well-suited for realistic workload environments.

End-to-End Index Build Time Comparison As summarized in Table 5, we compare the end-to-end index build time from scratch for three methods—the original `pgvector` (i.e., `pgv-orig`), DiskANN, and our holistic approach (i.e., `pgv-ours`), which incorporates `pgv-async-iou` ingestion, full PCA-based reordering, and locality-aware insertions. To reflect realistic ingestion scenarios for disk-resident indexes, all systems were evaluated under the same memory and high-concurrency settings to fully utilize available CPU cores and SSD bandwidth. Although DiskANN avoids significant I/O during index construction by relying on static in-memory sharding, we ensured a fair comparison by providing sufficient memory and evaluating DiskANN as a single in-memory shard. On the other hand, the `pgvector`-based methods, including our holistic approach, were tested under disk-bound conditions in which part of the data (5% for Deep-1M and C4-100K, and 50K vectors for the other datasets) was ingested through an I/O-intensive on-disk phase using a 95% memory buffer. Even though DiskANN was given a favorable setup, `pgv-ours` achieves more than 10× faster index construction on high-dimensional datasets such as C4-100K, mainly because DiskANN incurs substantial overhead when calculating centroids for each subspace during the quantization process. At larger scales such as Deep-100M, `pgv-ours` still achieves a 3.6× speedup, demonstrating its scalability for large-scale vector workloads.



(a) C4 5M (query: random) (b) C4 5M (query: MMLU) (c) Deep 1M (query: random) (d) Deep 100M (query: random)

Figure 11: Search Performance Comparison with DiskANN

Table 5: End-to-End Index build time from scratch (seconds)

Dataset	C4 100K	C4 1M	C4 5M	Deep 1M	Deep 10M	Deep 100M
DiskANN	823	2,451	7,353	416	2,520	13,821
pgv-orig	103	734	4,035	147	787	6,017
pgv-ours	78	454	3,534	77	513	3,887

8 RELATED WORK

Disk-based Vector ANN Disk-based vector databases address the limitations of DRAM-bound indexing for large-scale vector search. Systems such as DiskANN [21], ScaNN [17], and SPANN [11] store compressed indices in memory and retrieve raw vectors from disk, using graph- or tree-based quantized indexing to enable high-recall search under low memory budgets. However, their static designs limit support for dynamic updates. DiskANN++[33] improves I/O efficiency by optimizing prefetching and node traversal, reusing pages and colocating related nodes. In parallel, FreshDiskANN[39] supports dynamic updates through periodic merges and index freshness guarantees. While extending DiskANN, both still rely on quantization and require significant engineering. In contrast, our work targets dynamic, disk-resident vector databases using uncompressed graph-based indices like HNSW. Rather than modifying index structures or relying on quantization, we focus on optimizing graph traversal patterns and query execution to fully leverage SSD parallelism and cache locality within a DBMS-integrated environment.

In a related direction, Neos [20] presents a user-space vector buffer engine for real-time search over unindexed streaming data. It performs GPU-based brute-force search and bypasses the Linux I/O stack via SPDK. This enables low-latency retrieval for ephemeral or streaming workloads. In contrast, our work targets disk-resident HNSW indices, optimizing I/O by leveraging `io_uring` for asynchronous, parallel I/O scheduling.

Insertion Reordering In traditional indexes like B+-Tree, sorted insertions improve efficiency by reducing node splits and page reads, while in HNSW, insertion order influences traversal patterns and page access locality. Nodes with similar neighbors often access overlapping pages, indicating that reordering insertions could improve page locality and reduce I/O costs. Supporting this idea, RUMMY [47] showed that query reordering reduces cache misses and improves execution efficiency. Similarly, in HNSW, grouping insertions with similar traversal paths can improve cache locality and reduce redundant I/O, thereby accelerating index construction.

Index Partitioning Graph-based indexing in disk-based environments needs to consider neighbor relationships to maintain locality. Starling [43] and DiskANN++ [33] propose block layouts that colocate highly connected nodes within the same storage blocks. But their static designs degrade over time as insertions cause fragmentation, often requiring costly repartitioning. To address this, we propose a locality-aware insertion strategy that dynamically places nodes to preserve neighbor proximity and sustain query performance without excessive overhead.

9 CONCLUSION

This paper addresses key performance challenges in disk-based vector databases when using modern SSDs, and introduces a comprehensive set of optimization techniques, which includes leveraging flash memory parallelism, clustering vector embeddings, and optimizing embedding order, to enhance efficiency. Our implementation within `pgvector` demonstrates that each method independently achieves substantial performance gains, while their combined application leads to even greater improvements.

Experimental evaluations confirm that our approach significantly reduces I/O overhead, maximizes SSD parallelism, and enhances query throughput. These findings underscore the importance of optimizing storage access patterns and leveraging modern hardware capabilities for scalable vector search. Future work includes further refining SSD-aware indexing mechanisms, and extending these optimizations to other vector database frameworks. We believe these advancements will contribute to the next generation of high-performance, SSD-based vector retrieval systems.

ACKNOWLEDGMENTS

This work was supported in part by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant (MSIT) (No. RS-2024-00454666, Developing a Vector DB for Long-Term Memory Storage of Hyperscale Al Models, NO.RS-2021-11211343, Artificial Intelligence Graduate School Program (Seoul National University)); by an Industry- Academia collaborative project with Dnotitia (Project No. 2025-SNU-001); by the National Research Foundation of Korea (NRF) grant (MSIT) (RS-2024-00414981); by the Korea Institute of Science and Technology Information (KISTI) (No. K25L1M1C1), aimed at developing KONI (KISTI Open Neural Intelligence), a LLM specialized in science and technology; by Samsung Electronics; and by research facilities provided by the Samsung Memory Research Center (SMRC); by Creative-Pioneering Researchers Program through Seoul National University. J. Do is with ASRI, Seoul National University.

REFERENCES

[1] http://pinecone.io. Accessed: 2025-03-01.

[2] http://trychroma.com. Accessed: 2025-03-01.

[3] http://milvus.io. Accessed: 2025-03-01.

[4] H. Amiri and A. Shahbahrami. Simd programming using intel vector extensions. *J. Parallel Distrib. Comput.*, 135(C):83–100, Jan. 2020.

[5] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. Dbpedia: A nucleus for a web of open data. In *international semantic web conference*, pages 722–735. Springer, 2007.

[6] J. Axboe. FIO (Flexible IO Tester). https://github.com/axboe/fio. Accessed: 2025-03-01.

[7] J. Axboe. liburing (library providing helpers for the linux kernel io_uring support). https://github.com/axboe/liburing. Accessed: 2025-03-01.

[8] J. Bai and S. B. et al. Qwen technical report, 2023.

[9] E. Bingham and H. Mannila. Random projection in dimensionality reduction: applications to image and text data. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 245–250, 2001.

[10] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 266–277, 2011.

[11] Q. Chen, B. Zhao, H. Wang, M. Li, C. Liu, Z. Li, M. Yang, and J. Wang. Spann: Highly-efficient billion-scale approximate nearest neighborhood search. *Advances in Neural Information Processing Systems*, 34:5199–5212, 2021.

[12] D. Didona, J. Pfefferle, N. Ioannou, B. Metzler, and A. Trivedi. Understanding modern storage apis: a systematic study of libaio, spdk, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, pages 120–127, 2022.

[13] J. Dodge, M. Sap, A. Marasović, W. Agnew, G. Ilharco, D. Groeneveld, M. Mitchell, and M. Gardner. Documenting large webtext corpora: A case study on the colossal clean crawled corpus. *arXiv preprint arXiv:2104.08758*, 2021.

[14] S. Electronics. Samsung memory research center. https://smrc.biz.samsung.com/, 2024.

[15] G. T. et al. Gemma: Open models based on gemini research and technology, 2024.

[16] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[17] R. Guo, P. Sun, E. Lindgren, Q. Geng, D. Simcha, F. Chern, and S. Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020.

[18] N. Halko, P.-G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011.

[19] D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

[20] Y. Huang, X. Fan, S. Yan, and C. Weng. Neos: A nvme-gpus direct vector service buffer in user space. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 3767–3781, 2024.

[21] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems*, 32, 2019.

[22] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed. Mistral 7b, 2023.

[23] M. Jung and M. T. Kandemir. Sprinkler: Maximizing resource utilization in many-chip solid state disks. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 524–535, 2014.

[24] M. Jung, E. H. Wilson, and M. Kandemir. Physically addressed queueing (paq): Improving parallelism in solid state disks. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 404–415, 2012.

[25] A. Kane. pgvector (Open-source vector similarity search for Postgres). https://github.com/pgvector/pgvector. Accessed: 2025-03-01.

[26] S. T. Leutenegger and D. Dias. A modeling study of the tpc-c benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, page 22–31, New York, NY, USA, 1993. Association for Computing Machinery.

[27] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft coco: Common objects in context. In D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, editors, *Computer Vision – ECCV 2014*, pages 740–755, Cham, 2014. Springer International Publishing.

[28] S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[29] Y. A. Malkov and D. 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, 2018.

[30] Microsoft. DiskANN: Fast, Accurate Billion-Scale Nearest Neighbor Search on a Single Node. https://github.com/microsoft/DiskANN. Accessed: 2025-03-01.

[31] D. Newman. Bag of Words. UCI Machine Learning Repository, 2008. DOI: https://doi.org/10.24432/C5ZG6P.

[32] A. Y. Ng, M. I. Jordan, and Y. Weiss. On spectral clustering: analysis and an algorithm. In *Proceedings of the 15th International Conference on Neural Information Processing Systems: Natural and Synthetic*, NIPS'01, page 849–856, Cambridge, MA, USA, 2001. MIT Press.

[33] J. Ni, X. Xu, Y. Wang, C. Li, J. Yao, S. Xiao, and X. Zhang. Diskann++: Efficient page-based search over isomorphic mapped graph index using query-sensitivity entry vertex. *arXiv preprint arXiv:2310.00402*, 2023.

[34] J. J. Pan, J. Wang, and G. Li. Survey of vector database management systems. *The VLDB Journal*, 33(5):1591–1615, 2024.

[35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[36] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[37] PostgreSQL. The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/. Accessed: 2025-03-01.

[38] D. A. Reynolds et al. Gaussian mixture models. *Encyclopedia of biometrics*, 741(659-663), 2009.

[39] A. Singh, S. J. Subramanya, R. Krishnaswamy, and H. V. Simhadri. Freshdiskann: A fast and accurate graph-based ann index for streaming similarity search. *arXiv preprint arXiv:2105.09613*, 2021.

[40] The Linux Kernel Developers. *aio(7) Linux User's Manual*, 6.10 edition, March 2025. Accessed: 2025-03-01.

[41] The Linux Kernel Developers. *epoll(7) Linux User's Manual*, 6.10 edition, March 2025. Accessed: 2025-03-01.

[42] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample. Llama: Open and efficient foundation language models, 2023.

[43] M. Wang, W. Xu, X. Yi, S. Wu, Z. Peng, X. Ke, Y. Gao, X. Xu, R. Guo, and C. Xie. Starling: An i/o-efficient disk-resident graph index framework for high-dimensional vector similarity search on data segment. *Proc. ACM Manag. Data*, 2(1), Mar. 2024.

[44] D. Xu, I. W. Tsang, and Y. Zhang. Online product quantization. *IEEE Transactions on Knowledge and Data Engineering*, 30(11):2185–2198, 2018.

[45] H. Xu, M. D. Manohar, P. A. Bernstein, B. Chandramouli, R. Wen, and H. V. Simhadri. In-place updates of a graph index for streaming approximate nearest neighbor search, 2025.

[46] A. B. Yandex and V. Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2055–2063, 2016.

[47] Z. Zhang, F. Liu, G. Huang, X. Liu, and X. Jin. Fast vector query processing for large datasets beyond GPU memory with reordered pipelining. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, NSDI'24, USA, 2024. USENIX Association.

[48] T. Zhong, Z. Liu, Y. Pan, Y. Zhang, Y. Zhou, S. Liang, Z. Wu, Y. Lyu, P. Shu, X. Yu, et al. Evaluation of openai o1: Opportunities and challenges of agi. *arXiv preprint arXiv:2409.18486*, 2024.